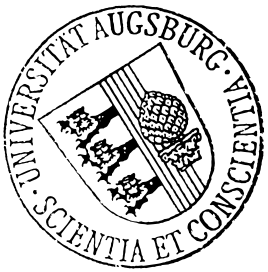
Algebraic Calculi for Separation Logic

Dissertation zur Erlangung des Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat)

an der Fakultät für Angewandte Informatik
der Universität Augsburg



vorgelegt von

Han Hing Dang

2014

Gutachter:
Erstgutachter: Prof. Dr. Bernhard Möller
Zweitgutachter: Prof. Dr. Bernhard Bauer

Tag der mündlichen Prüfung:
09. Dezember 2014

Love is not finding someone to live with.
It's finding someone you can't live without.
Rafael Ortiz

To my fiancée Olga

Contents

Preamble **1**

1 Introduction **5**
1.1 Motivation 5
1.2 Separation Logic 7
1.3 Algebras for Pointer Structures 9
1.4 Contributions and Organisation 10

2 Separation Logic — An Overview **12**
2.1 A Storage Model and Spatial Assertions 12
2.2 Program Constructs for Resource Manipulation 16
2.3 The Frame Rule 20

3 Algebraic Spatial Assertions **23**
3.1 A Denotational Model for Assertions 23
3.1.1 Related Work: BI Algebras 31
3.2 Characterising Behaviour Abstractly 32
3.2.1 Intuitionistic Assertions 33
3.2.2 Resource Independence 36
3.2.3 Preciseness 40
3.2.4 Full Allocation 42
3.2.5 Supported Assertions 44
3.3 Relationship to Separation Algebras 51

4 Relational Separation **57**
4.1 Interpreting Commands Relationally 57
4.2 On Partial and Total Correctness 61
4.3 Abstracting Modularity 67
4.3.1 A Pointfree Frame Property 72
4.3.2 Resource Preservation 76

4.3.3 A Calculational Proof of the Frame Rule . . . 78
4.3.4 Related Algebraic Approaches . . . 81
4.4 Applications to Concurrency . . . 85
4.4.1 Relations and Concurrent Separation Logic . . . 85
4.4.2 Disjoint Concurrency . . . 94
4.4.3 Concurrent Kleene Algebras . . . 98
4.5 Pointfree Dynamic Frames . . . 109
4.5.1 Abstracting Dynamic Frames . . . 110
4.5.2 Locality and Frame Accumulation . . . 116

5 Transitive Separation Logic **123**
5.1 The Algebraic Foundation . . . 123
5.2 A Stronger Notion of Separation . . . 127
5.3 An Algebra of Linked Structures . . . 132
5.4 Structural Properties of Linked Structures . . . 134
5.5 Assertions and Program Commands . . . 139
5.6 Inference Rules . . . 143
5.6.1 Selector Assignments . . . 143
5.6.2 Frame Rules . . . 144
5.7 Verification Examples . . . 148
5.7.1 List Reversal . . . 148
5.7.2 Tree Rotation . . . 150
5.8 A Treatment of Overlaid Data Structures . . . 154

6 Conclusion **161**
6.1 Summary . . . 161
6.2 Future Work . . . 162

A Deferred Proofs and Properties **165**
A.1 Deferred Proofs . . . 165
A.2 Further Properties of the Assertion Calculus . . . 178
A.3 Deferred Figures . . . 184

Bibliography **185**

List of Figures **199**

Index **200**

Curriculum Vitae **203**

Preamble

Übersicht

Ein bedeutendes Forschungsthema für die moderne Softwaretechnik ist die Entwicklung von formalen Methoden, die Korrektheit von Computerprogrammen bzgl. ihrer Spezifikation sicherstellen. Diverse Verfahren wurden innerhalb der letzten Jahrzehnte entwickelt, speziell im Fachgebiet der logischen Methoden. Eine der einflussreichsten und bekanntesten Methodiken aus diesem Bereich ist die *Separationslogik*. Sie hat sich aus der Hoare-Logik entwickelt, um speziell die Beweisführung auf Programmen mit einer Vielzahl an Referenzen auf dynamisch reservierten Speicher zu vereinfachen. Durch spezielle Mechanismen erlaubt sie einfache Formeln zur Charakterisierung der Formen und Strukturen von Datentypen. Insbesondere hat sich diese Logik durch die Möglichkeit einer kompositionellen Konstruktion von Korrektheitsbeweisen als skalierbar erwiesen, speziell für komplexeren Programmcode. Während der letzten Jahre wurde eine Vielzahl von Ausprägungen in diesem Forschungsbereich geschaffen, die sich von Anwendungen für Nebenläufigkeit bis hin zur Mechanisierung und programmgestützten Verifikation von imperativen und objektorientierten Programmen erstrecken.

Jede dieser anwendungsspezifisch entwickelten Separationslogiken erweitert den ursprünglichen Kern, der skalierbare Beweisführung ermöglicht, um eine spezielle Semantik und syntaktische Ausdrücke. Jedoch sind die meisten dieser Kalküle sehr komplex und nicht weitreichend anwendbar oder sie verwenden allgemeingültige Abstraktionen, die schwer zu verstehen sind und nur mühsam von Nicht-Experten gehandhabt werden können. Im Vergleich dazu bieten algebraische Techniken einen balancierten Mittelweg für beide Probleme. Einerseits sind sie ausreichend abstrakt und allgemein um Verhalten zu erfassen und darzustellen. Andererseits vereinfachen sie Beweise durch einfache und (un)gleichungsbasierte Formeln, die Herleitungen von nicht-trivialen Konsequenzen und Eigenschaften ermöglichen. Das Ziel der vorliegenden Dissertation besteht aus der Entwicklung von algebraischen Kalkülen für eine uniforme Darstellung und Abstraktion von Verhalten in Separationslogiken. Dies er-

möglicht im Speziellen generelle Resultate einer Theorie auf eine andere zu übertragen. Darüber hinaus können durch die Verwendungen einfacher Formeln, auch auf abstrakter Ebene, Programmwerkzeuge zur Unterstützung und Steuerung der Entwicklung weiterer Theorien verwendet werden.

Abstract

A major research topic for the discipline of software engineering is the development of formal methods that ensure correctness of computer programs w.r.t. their specifications. Various approaches have been developed over the last decades, especially in the field of logical methods. One of the most influential and popular methodologies in this area is *separation logic*. It has evolved from Hoare logic as a treatment that facilitates reasoning about programs that massively work with references to dynamically allocated storage. Due to special mechanisms it allows simple formulas for the characterisation of shapes and structures of data types. Moreover, it has proven to be scalable by enabling a compositional construction of correctness proofs in particular for large program code. During the last years various developments in this research area have been established ranging from applications within concurrency to mechanisation and tool-supported verification of imperative and object-oriented programs.

Each application-specific separation logic introduces special syntax and semantics on top of the original core that enables scalable reasoning. However, most of the calculi are very complex and not widely applicable, or they involve general abstractions that are difficult to understand and handle for non-experts. By contrast, algebraic techniques provide a balanced compromise for both problems. On the one hand they are abstract and general enough to capture and represent behaviour in a concise and simple way. On the other hand they facilitate reasoning by formulas in an (in)equational style that allow derivations of non-trivial consequences and properties. The aim of the present thesis is to develop algebraic calculi for a uniform representation and abstraction of behaviour in separation logics. This yields in particular the possibility of transferring general results between various separation logical theories. Moreover, due to simple formulas expressed within first-order logic they also enable at the abstract level a tool support for developing further theories.

Acknowledgement

First of all, I am most grateful to my supervisor Prof. Dr. Bernhard Möller for giving me the possibility to write a doctoral thesis. Without his support, motivation and encouragements during the years I would never have finished this work. Moreover, I want to thank Prof. Dr. Bernhard Bauer for reviewing this thesis.

I am also very grateful to Dr. Peter Höfner who has already supported me at the time when I finished my diploma thesis. His never ending variety of ideas in discussions always impressed me and enormously influenced my thinking in developing solutions for research-related problems.

Moreover, I would like to thank my colleagues Dr. Markus Endres, Roland Glück, Dr. Alfons Huhn, Prof. Dr. Werner Kießling, Dominik Köppl, Dr. Martin Müller, Patrick Roocks and Florian Wenzel for an enjoyable atmosphere during the past years at the Universität Augsburg and in particular Andreas Zelend and Alba vom Wolfschlag for enjoyable conversations. I also thank Dr. Georg Struth for fruitful discussions and all reviewers of the RAMiCS, MPC conferences, the ATE, PAAR workshops and of the JLAP, JLAMP and SCP journals for helpful and inspiring comments.

For financial support I thank Prof. Dr. Bernhard Möller for providing me with teaching assistant jobs during my first years. I would also like to thank Prof. Dr. Werner Kießling and Prof. Dr. Dirk Hachenberger for bridging fundings with further teaching assistant jobs. Moreover, I gratefully acknowledge the German research foundation (DFG) for funding a position and especially many thanks to Prof. Dr. Bernhard Möller and Dr. Peter Höfner for their efforts in writing all of the proposals over the last years. Finally, I am grateful to Sir Tony Hoare for the cooperation within the DFG-project AlgSep.

Also I am most grateful to my family for any support during the last years. In particular, many thanks to my brothers Chi Tai and Han Kie for the endless discussions about any computer science related topics and my sister Anna for always giving me advice in any matter. Moreover, I am deeply grateful to my fiancée Olga for always supporting me in any decisions I made and being there for me whenever I needed someone in my life.

Finally, many thanks to my friends and all people who supported me during the past years.

Chapter 1

Introduction

Separation Logic was developed to facilitate reasoning about shared mutable data structures in a Hoare logic style. It comes with suitable operations and spatial predicates that ensure for frequently used data structures central correctness properties as the absence of sharing resources. There exists also a variety of algebraic approaches that reflect central concepts for the treatment of such data structures. In this section we provide some historical background on separation logic and algebraic approaches for pointer structures. Moreover, we give a short overview on recent developments and conclude by summarising the structure and contributions of this thesis.

1.1 Motivation

Many formal methods have been developed during the past decades to ensure correctness of programs that heavily work with pointers, i.e., references to resources of a program. This has been proven to be a difficult and tedious task, especially with logical calculi by Hoare and Dijkstra in their original forms [Hoa69, Dij76]. A reason for this is that these treatments do not provide adequate and general enough constructs for dealing with complex data structures. The problem therefore is that certain properties or invariants have to be defined in a fashion that is difficult to understand and read. This in turn makes the lengthy correctness proofs less reliable and the whole approach usable only for experts, i.e., the minority of users.

Hence, Reynolds, O'Hearn and others introduced an extension of such calculi, called

separation logic [Rey02], that provides operators to facilitate the task of specifying the mentioned properties and invariants of data structures. The speciality of this logic is a connective, called *separating conjunction*, that ensures disjointness of sets of resources. This has the advantage that the resources of the disjoint sets cannot be aliases of each other. In combination with recursively defined predicates it allows relatively simple characterisations of shared mutable data structures such as singly and doubly-linked lists or tree structures. In addition to that the logic also validates a special inference rule called the *frame rule*, which allows under certain assumptions local and modular reasoning about programs by focusing on relevant parts of the state space. This makes the approach more scalable and hence also applicable for tackling large programs by compositionally verifying procedures on smaller parts of storage and then obtaining a global proof of the program by reassembling the proofs of the parts.

Nowadays there exists a lot of research around separation logic, resulting in a multitude of logical calculi (see e.g., [Par10]) for particular applications ranging from information hiding [ORY09] to concurrency reasoning [O'H07] and rely/guarantee settings [VP07]. All of these treatments include the basic concepts of separation. Moreover, a variety of theorem proving tools on a decidable fragment of separation logic has been developed for automating the logic and verification tasks [BCO06, JP08, Tue08]. A general disadvantage of most approaches is that each calculus and corresponding theorem prover has to be developed anew, although their foundations and cores are the same. This development is cumbersome, expensive and time-consuming. In particular, the knowledge of experts is often required for introducing special behaviour in the setting. This can be facilitated by the abstraction from irrelevant details and concentrating on the foundations that establish the advantages and characteristics of separation logic.

Algebraic techniques have proved to be adequate for the abstraction of logical calculi. The abstract and calculational proofs enable formal reasoning using simple (in)equational laws as known from school algebra. Such laws can be used to describe the main core of all separation logic-based calculi and moreover enable the derivation of general and commonly used properties and inference rules. We develop such abstract and general algebraic calculi where one can compositionally enrich the basic setting with additional axioms that include special behaviour of various forms of separation logic calculi. Thus, the algebraic setting represents a compact and uniform representation of such. We provide, in particular, abstract and general formulations for the assertion language of separation logic which denote frequently reused parts. Moreover we characterise in a relational and pointfree style the local behaviour that establishes modularity of that approach, also in a concurrent environment. Moreover, due to abstractness we can relate the core of separation logic also to the theory of dynamic frames that is basically inspired by the concepts of separation. Using the

established formalisations we develop an extension to separation logic that further facilitates reasoning within graph structures by introducing several new operators.

A final advantage that comes with an algebraic treatment is that the obtained laws can directly be fed into existing fully automated theorem proving systems as done e.g., in [HS07, HS08, DH08]. This allows a tool-supported and tool-guided development of various separation-logical calculi without any need to construct proof systems for every special problem domain. In particular, this approach makes use of the stepwise evolving power of general-purpose theorem provers.

1.2 Separation Logic

The central concepts and ideas to keep resources of a program distinct appeared first in Burstall's work [Bur72] in 1972. According to [Rey09] these represented the first steps towards separation logic. A sound instance of that logic was introduced independently in 1999 by the authors Ishtiaq and O'Hearn in [IO01] and Reynolds in [Rey00]. In their works an intuitionistic version of the logic was developed that provides assertions with a monotonicity property in the following sense: if an assertion holds for some parts of the dynamically allocatable storage then it is also valid for any larger storage. The key concept of both approaches was a "spatial conjunction" on assertions for expressing separation between memory regions. Concretely, for arbitrary assertions p and q their *separating conjunction* $p * q$ asserts that p and q both hold, but each for a separate part of the storage.

In [IO01], Ishtiaq and O'Hearn also developed a variant within classical logic which is more expressive than the intuitionistic version. More concretely, their work does not incorporate the mentioned monotonicity property. In particular the starting point for their assertion language was another theoretical foundation called the *logic of bunched implications*, abbreviated by **BI** [OP99, Pym02]. This early approach was developed by O'Hearn and Pym and represented a logical proof system that also included the ideas for an abstract treatment of resources. In [OP99] a Kripke semantics for current separation logic assertions was provided that described the intuition for the separating conjunction and its adjoint, the *separating implication* alias the magic wand operation.

Building on this semantic foundation O'Hearn and others continued to develop another important ingredient of separation logic, called the *frame rule* [ORY01]. That special inference rule includes the concepts of separation and allows, in some circumstances, local reasoning about changing storage without affecting disjoint portions. This expresses the main power of separation logic as correctness proofs become scalable. A semantic foundation for this inference rule has been established in [YO02] yielding a denotational model for separation logic. Finally, the basic version of the

logic was presented in [Rey02] and extended by Reynolds with a command language that allows altering separate ranges and includes pointer arithmetic.

Starting from this, the logic had an immense influence on formal methods for reasoning about program correctness. In [Yan01, Yan07], an algorithm that is frequently used for garbage collectors is treated within separation logical approaches. The algorithm is called the *Schorr-Waite graph marking* and has the advantage that it only requires an extra bit per node to identify marked nodes [SW67]. A variant of separation logic that presents a correctness of a copying garbage collector can be found in [TSBR08]. Moreover, separation logic has been extended with proof rules that are suitable for information hiding in [ORY09]. As another application for the logic it has been adapted to object-orientation [PB05] coping with JAVA-like classes and procedures while maintaining modularity.

Further research considers separation logic and concurrency. First ideas to this have been developed in [O'H07] by O'Hearn, resulting in *concurrent separation logic*. It was used as a formal method to reasoning about concurrent programs that massively involve pointers. A semantics to this approach that proved soundness of that logic has been introduced by Brookes [Bro07]. A further proof that validates soundness by an operational semantics was developed in [Vaf11]. A concrete verification of a non-blocking stack in a concurrent setting that used the special proof rules of concurrent separation logic can be found in [PBO07]. There also exists another approach to verifying concurrent algorithms by so-called rely/guarantee techniques [CJ00, VP07]. This setting facilitates reasoning about interference by providing adequate proof rules and conditions under which assertions remain stable under certain interference, i.e., guarantee some behaviour. For this there exist also variants of separation logic that include the concept of permissions [BCY06].

Moreover, also at the data structure level there exists a variety of treatments yielding more suitable operators for reasoning about sharing [WBO08, HV13]. A modal extension to verify data-parallel pointer programs has been considered [Nis06]. Furthermore, a separation logic that copes with low-level programs has been introduced [TKN07]. Moreover, for automating the verification of program properties a multitude of extensions has been considered [CS10], also incorporating aspects of concurrency or enabling machine-supported verification, e.g., in tools like SmallFoot [BCO06] which is implemented on a decidable fragment of separation logic [BCO05], or the VeriFast program verifier [JP08]. In addition to this also higher-order logic theorem proving tools such as Isabelle/HOL have been combined with separation logic [Tue08]. Further research on automation considered shape analysis methodologies [YLB+08, CDOY09b] that in particular allowed the extraction of specifications and preconditions by the source code of a hardware driver and system code [CDOY09a].

A more theoretical view to extract the core behaviour of separation logical calculi was provided in various other works. A first comprehensive and useful abstraction is currently being explored in [DYBG+13] which provides a formal foundation and additional ingredients to obtain several separation logical calculi. Similar generalised approaches to this that are used to capture a wide range of models of separation logic was developed in the treatment of local actions and abstract separation logic [COY07]. Moreover, relationships to other frameworks such as the *theory of dynamic frames* has been discussed, since that approach was developed to tackle similar problems as separation logic does [SJP09].

1.3 Algebras for Pointer Structures

Early approaches on an algebraic treatment of pointer structures have been investigated from 1990 on by Möller [Möl92, Möl93a, Möl93b]. An algebraic foundation for pointer structures was introduced that already allowed the characterisation of frequently required properties like the absence of cycles or disjointness of the set of reachable nodes from a designated root node. The latter property corresponds closely to the central concept of separation logic assertions that guarantee by separating conjunction spatial disjointness of sets of resources. This allowed a calculational verification of algorithms on lists like their concatenation or reversal [Möl97]. Further investigations on this problem field led to observation on more complex data structures as trees, forests and particularly cyclic lists [Möl99a] yielding concepts to describe updates on pointer structures along specified links and the characterisation of sharing patterns and their exclusion.

Building on this algebraic approach, Ehm developed in 2003 a formal treatment of pointer structures called *pointer Kleene algebra* based on the algebraic structure of Kleene algebras [Ehm03, Ehm04]. These structures come with a special operation for finite iteration called the *Kleene star* and were introduced to model the theory of regular events. They have been extensively studied by Conway [Con71] in 1971, resulting in various axiomatisations of Kleene algebras based on quantales, which are a special case of idempotent semirings. In the case of pointer structures iteration is used to abstractly model reachability along arbitrarily many links. The approach of Ehm also includes elements of the theory of L-fuzzy relations, i.e., Goguen categories (e.g., [Win07]) to introduce labels on links and operations to extend the definitions of reachability on such abstract structures. Moreover, it has been shown that the algebraic treatment also allowed a derivational approach for obtaining correctness preserving functional definitions of pointer algorithms [Ehm01]. The reverse direction for a verification purpose in sense of Hoare logics has also been sketched in [Ehm03].

There are algebraic approaches for the propositional fragment of Hoare logics [Koz00,

MS06a] and the wp-calculus of Dijkstra [MS06b] that also consider Kleene algebras and quantales. In particular, they have been used in various applications ranging from concurrency control [Coh94, HMSW09a, HMSW09b] to program analysis [KP00] and semantics [MHS06]. The algebraic approach achieved several goals. The view became more abstract, which led to a considerable reduction of detail and hence allowed simpler and more concise proofs. On some occasions also additional precision was gained. Furthermore, the algebraic abstraction places the considered theories into a more general context and therefore allows re-use of a large body of existing results.

The used algebraic structures, i.e., Kleene algebras and idempotent semirings, are formulated in pure first-order logic. This further enables the use of off-the-shelf automated theorem provers for verifying properties at the more abstract level [HS07, HSS08]. A lot of feasibility and case studies have been investigated during the recent years [Str07, Höf08], particularly for the case of pointer Kleene algebra [DM11]. Moreover, various theorem proving systems that can be found within the TPTP Library [SS98] have been evaluated with the mentioned algebraic structures. As one result of this, Prover9 [McC05] turned out to be the most adequate system for automating these tasks [DH08]. Most of the input files can be found at the web page [Höf] for the interested reader. However, the case of quantales is slightly different as it comes with axioms not expressive within first-order logic. An encoding of an axiomatisation and some automated proofs of basic properties within higher-order logic can be found in [DH12] with less promising results with nowadays standard systems. Newer approaches on this topic use semi-automated proof assistants like Isabelle/HOL [AS12, ASW13b] or Coq [BP12]. An extensive amount of proofs using Isabelle/HOL can be found in [ASW13a].

1.4 Contributions and Organisation

The contribution of this thesis consists of three parts. We developed an abstraction of the spatial assertion of separation logic based on quantales. For this we defined a set-based variant of the separating conjunction that enabled simple algebraic proofs of main properties. Moreover, by the abstraction to quantales this further allowed pointfree inequational characterisations of assertion classes. The abstract developments also allow the transfer of the gained results to other separation logical theories. As the second main contribution we developed a relational calculus to model the effects and behaviour of separation logic that guarantee its modularity and scalability in program proofs. This allowed further formulations for other separation logical calculi in a sequential and also concurrent setting. The last contribution that we present is an algebraic extension of separation logic for a more suitable treatment of pointer or linked object structures. This approach significantly allows simple correctness proofs

of algorithms on linked data structures that split into one part guaranteeing preservation of structural invariants and another preserving functional correctness.

The thesis is organised as follows:

Chapter 2 gives an overview of separation logic in its classical form. First, we provide a standard storage model and main concepts of the assertion language. Moreover, we present the programming layer of the logic itself and introduce definitions that establish modularity within that approach.

In **Chapter 3** we continue with a denotational model for the assertions of separation logic based on sets of states. We further abstract this structure to general quantales which allows the exclusion of irrelevant details of separation logic assertions. As our first contribution we provide completely pointfree characterisations of well-known and frequently used assertion classes. This yields fully algebraic and abstract proofs of central properties in a calculational style.

The second contribution of this thesis can be found in **Chapter 4**. There, we provide a relational calculus extended to cope with separation. In particular, we give formulations to include the fault-avoiding triple definition of separation logic into the pointfree setting and developed characterisations of central properties and definitions to establish soundness of the frame rule. Finally, we give a concise and algebraic proof of that inference rule and extended the formulations to incorporate also concurrency proof rules. As a final step for this chapter we provide relationships of the relational treatment to other similar approaches as, e.g., concurrent Kleene algebras in the case of concurrency and the dynamic frames theory as another approach that involves framing.

Chapter 5 represents the third contribution of this thesis and gives an extension to separation logic at its data structure level. In this chapter we replace the resources of separation logic by elements of a modal Kleene algebra that abstractly capture pointer or linked structures. By this we give definitions of operations and predicates that allow simple proofs of preservation of tree-like structures. Moreover, we present as case studies for that approach correctness proofs of algorithms for lists, trees and in particular threaded trees that involve both data structures.

Finally, **Chapter 6** provides a summary of this thesis and gives some open questions for future work.

In the **Appendix** one can find all deferred proofs and properties for the interested readers.

Chapter 2

Separation Logic — A Short Overview

In this chapter we give basic definitions of the standard approach of separation logic that was introduced in [Rey02]. We provide a standard storage model on which the separation logical assertions are evaluated. Moreover, we give all standard definitions of the spatial assertions and present some simple examples to demonstrate the main concepts for establishing correctness of frequently used data structures. As another important concept we give operational semantics for the program commands of that logic and provide formulations that entail scalability of the whole approach.

2.1 A Storage Model and Spatial Assertions

As already mentioned, separation logic is an extension of Hoare logic and, besides reasoning about explicitly named program variables, it comes with additional new connectives for a flexible treatment of dynamically allocated storage. For this extension, a program state in separation logic consists of a *store* and a *heap* component. In contrast, plain Hoare logic states only involve a store, since just values of used program variables have to be remembered. In the remainder we consistently write s for stores and h for heaps.

For a formal model of the underlying storage we first provide some definitions. In the standard approach one defines values and addresses as integers, stores and heaps

as partial functions from variables or addresses to values and states as pairs of stores and heaps:

$$\begin{aligned} \mathit{Values} &= \mathbb{Z}\,, \\ \{\mathsf{nil}\} \dot\cup \mathit{Addresses} &\subseteq \mathit{Values}\,, \\ \mathit{Stores} &= V \rightsquigarrow \mathit{Values}\,, \\ \mathit{Heaps} &= \bigcup_{A} (A \rightsquigarrow \mathit{Values})\,, \quad (A \subseteq \mathit{Addresses},\ A \text{ finite}) \\ \mathit{States} &= \mathit{Stores} \times \mathit{Heaps}\,, \end{aligned}$$

where V denotes the set of program variables, $\dot\cup$ is with disjoint union on sets and $M \rightsquigarrow N$ means the set of partial functions between arbitrary sets M and N. The constant nil is handled as an improper reference like null in the imperative programming language $\mathtt{C}$. By the above definition, nil is not an address and hence heaps do not assign values to nil, which is a natural requirement. The domain of a relation modelling a partial function R is defined by

$$\mathit{dom}(R) =_{df} \{x : \exists\, y : (x, y) \in R\}\,.$$

More concretely, the domain of a store $\mathit{dom}(s)$ denotes all variables currently used by a program while $\mathit{dom}(h)$ is the set of all allocated addresses on a heap h.

As in [Möl93b] and for later definitions of program commands we also need an *update* operator to model changes in stores and heaps. Let f_1 and f_2 be partial functions. Then we define

$$f_1 \,|\, f_2 =_{df} f_1 \cup \{(x, y) : (x, y) \in f_2 \wedge x \notin \mathit{dom}(f_1)\}\,. \tag{2.1}$$

By this, f_1 updates the partial function f_2 with all possible pairs (x, y) of f_1 in such a way that $f_1 \,|\, f_2$ is again a partial function. The domain of the right argument of $\cup$ is disjoint from that of f_1. In particular, $f_1 \,|\, f_2$ can be seen as an extension of f_1 to $\mathit{dom}(f_1) \cup \mathit{dom}(f_2)$. We abbreviate an update $\{(x, y)\} \,|\, f$ on a single variable or address by omitting the set-braces and write $(x, y) \,|\, f$ instead.

Now, *expressions* in separation logic are defined to be independent of the heap and hence only need the store component of a given state for their evaluation. This entails that their evaluation will not have any side effects. As in Hoare logic, they simply denote values or Boolean conditions. Syntactically, we distinguish *exp*-expressions which are arithmetical expressions over variables and values and *bexp*-expressions which are Boolean expressions, i.e., comparisons and true, false:

$$\begin{aligned} \mathit{var} &::= x \mid y \mid z \mid ... \\ \mathit{exp} &::= 0 \mid 1 \mid 2 \mid ... \mid \mathit{var} \mid \mathit{exp} \pm \mathit{exp} \mid ... \\ \mathit{bexp} &::= \mathsf{true} \mid \mathsf{false} \mid \mathit{exp} = \mathit{exp} \mid \mathit{exp} < \mathit{exp} \mid ... \end{aligned}$$

Assuming that all free variables of an expression e are contained in $dom(s)$, the semantics e^s of an expression e w.r.t. a store s is straightforward. For example, $\forall z \in \mathit{Values} : z^s = z$, $\mathsf{true}^s = \mathsf{true}$ or $\mathsf{false}^s = \mathsf{false}$.

As a next step, we define syntax and semantics of separation logic assertions. They extend the Hoare logic ones with additional constructs to make assumptions about the heap component of a state. Their syntax is defined by

$$\begin{aligned} \mathit{assert} ::= {} & \mathit{bexp} \mid \neg\, \mathit{assert} \mid \mathit{assert} \vee \mathit{assert} \mid \forall\, \mathit{var}.\, \mathit{assert} \mid \\ & \mathsf{emp} \mid \mathit{exp} \mapsto \mathit{exp} \mid \mathit{assert} * \mathit{assert} \mid \mathit{assert} \mathrel{-\!\!*} \mathit{assert}\,. \end{aligned}$$

The assertions in the upper row are known from predicate logic while the ones below can be used to express spatial properties about the heap. In the following we use the letters p, q and r for assertions. Note, the standard ones above can be supplemented by the logical connectives $\wedge$, $\rightarrow$ and $\exists$ that are defined, as usual, by $p \wedge q =_{df} \neg(\neg p \vee \neg q)$, $p \rightarrow q =_{df} \neg p \vee q$ and $\exists v : p =_{df} \neg \forall v : \neg p$.

The semantics of assertions is given by a relation $s, h \models p$ of *satisfaction*. Informally, $s, h \models p$ holds iff the state (s, h) satisfies the assertion p. The semantics is defined inductively as follows (cf. [Rey09]):

$$\begin{aligned}
s,h \models b &\Leftrightarrow_{df} b^s = \mathsf{true} \\
s,h \models \neg p &\Leftrightarrow_{df} s,h \not\models p \\
s,h \models p \vee q &\Leftrightarrow_{df} s,h \models p \quad \text{or} \quad s,h \models q \\
s,h \models \forall v :\ p &\Leftrightarrow_{df} \forall x \in \mathbb{Z} : (v,x) \,|\, s,h \models p \\
s,h \models \mathsf{emp} &\Leftrightarrow_{df} h = \emptyset \\
s,h \models e_1 \mapsto e_2 &\Leftrightarrow_{df} h = \{(\, e_1^s\,,\, e_2^s\,)\} \\
s,h \models p * q &\Leftrightarrow_{df} \exists h_1, h_2 \in \mathit{Heaps} : dom(h_1) \cap dom(h_2) = \emptyset \text{ and} \\
&\qquad h = h_1 \cup h_2 \text{ and } s,h_1 \models p \text{ and } s,h_2 \models q \\
s,h \models p \mathrel{-\!\!*} q &\Leftrightarrow_{df} \forall h' \in \mathit{Heaps} : (dom(h') \cap dom(h) = \emptyset \text{ and } s,h' \models p) \\
&\qquad \text{implies } s, h' \cup h \models q\,.
\end{aligned}$$

Here, b is a *bexp*-expression and e_1, e_2 are *exp*-expressions. As mentioned, the first four clauses do not consider the heap and are well known (e.g. [Hoa69]). The remaining lines express the meaning of the new constructs: emp ensures that the heap h is empty and hence contains no addressable cells. The assertion $e_1 \mapsto e_2$ characterises the heap of a state to contain exactly one cell at the address e_1^s with value e_2^s. For building up more complex heaps, the operator of *separating conjunction* $*$ is introduced. It can conversely also be interpreted as a connective that ensures properties on disjoint regions of the underlying heap. Finally, a state (s, h) satisfies the *separating implication* $p \mathrel{-\!\!*} q$ if h ensures that whenever it is extended with a disjoint heap h' with $(s, h) \models p$, the combined heap $h \cup h'$ needs to satisfy $(s, h \cup h') \models q$. An illustration of this can be found in Figure 2.1. This allows under some circumstances

the extraction of a disjoint subheap h' from the larger heap h which is useful for a variety of applications, e.g., in case of characterising unspecified sharing within data structures [HV13].

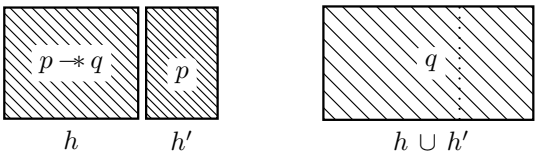


Figure 2.1: Illustration of separating implication.

There exists another special operation in separation logic although not mentioned in the classic literature [Rey02]. It is called *septraction* and denotes an existential version of the separating implication which quantifies over all subheaps h'. We will provide its concrete definition later in Section 3 where we also give an algebraic version of that operator. A concrete application for septraction can be found in [VP07] in concurrent contexts. It is used to characterise stability of assertions, i.e., preservation of validity under certain changes of resources by an environment or other threads.

As a next step we present some small examples with the new connectives in action and to better understand their usage. In particular, we demonstrate the effectiveness of separating conjunction for characterising commonly used data structures by the example of lists and trees. Following [Rey09], we start with a predicate definition for the former structure. First, we introduce some syntactic sugar by the assertion $\mathsf{i} \mapsto v_1, v_2$. It is a shorthand for $(\mathsf{i} \mapsto v_1) * (\mathsf{i}+1 \mapsto v_2)$ which characterises two adjacent heap cells starting at address v^s with contents v_1^s, v_2^s w.r.t. a store s.

Example 2.1.1 Lists can be structurally defined by an inductive predicate $\mathsf{list}\ \alpha\ \mathsf{i}$ where i denotes a program variable and α is used as an abstract sequence of values that represents the contents of the complete list. Assume ε denotes the empty word then

$$\begin{aligned} \mathsf{list}\ \varepsilon\ \mathsf{i} &\Leftrightarrow_{df} \mathsf{emp} \wedge \mathsf{i} = \mathsf{nil}\,, \\ \mathsf{list}\ (a \cdot \alpha)\ \mathsf{i} &\Leftrightarrow_{df} \exists \mathsf{j}.\ (\mathsf{i} \mapsto a, \mathsf{j}) * \mathsf{list}\ \alpha\ \mathsf{j}\,. \end{aligned}$$

The upper case describes an empty list. The sequence and the corresponding heap of the list predicate are empty. In particular, the variable i is required to hold the improper reference nil. The second case is more interesting as it characterises a non-empty list. In this, the head element asserted with $\mathsf{i} \mapsto a, \mathsf{j}$ of the list can be made visible. With this definition the value a is stored in the first cell while an anonymous address to the rest of the list is saved in the second cell denoted by the variable j.

Such addresses are generally realised in separation logic by existentially quantified variables in formulas. Note, that separating conjunction in this case implies that i and j can not hold the same address, i.e., they are not aliases. Moreover, one can easily use the following formula for sequences α, β

$$\mathsf{list}\ \alpha\ \mathsf{i} * \mathsf{list}\ \beta\ \mathsf{j}$$

to characterise two disjoint lists on the heap. By the usage of separating conjunction and the recursive definition of the predicate list one can show that both lists can not share some of their allocated heap cells. □

Example 2.1.2 Another example is given by the following definition that characterises the shape of a tree data structure. For representing the values of the data fields in a tree so-called S-expressions [Rey09] are used which we will not elaborate here. Conceptually the recursive definition of tree predicates is similar to that of lists:

$$\begin{aligned} \mathsf{tree}\ a\ \mathsf{i} &\Leftrightarrow_{df} \mathsf{emp} \wedge \mathsf{i} = a\,, \\ \mathsf{tree}\ (\tau_0 \cdot \tau_1)\ \mathsf{i} &\Leftrightarrow_{df} \exists \mathsf{i}_0, \mathsf{i}_1.\ \mathsf{i} \mapsto \mathsf{i}_0, \mathsf{i}_1 * (\mathsf{tree}\ \tau_0\ \mathsf{i}_0) * (\mathsf{tree}\ \tau_1\ \mathsf{i}_1)\,. \end{aligned}$$

The base case is represented by an empty heap where the data value of the tree is kept in the variable i. Compared to lists, the above base case in the definition of trees implies that at least some value a, not necessarily nil, is contained in any tree. The recursive case is similar as above for lists where $\mathsf{tree}\ \tau_i\ \mathsf{i}_\mathsf{i}$ represent the left and right subtrees of the larger tree. Again, it can be seen that by the assertion $\mathsf{tree}\ \tau_0\ \mathsf{i}_0 * \mathsf{tree}\ \tau_1\ \mathsf{i}_1$ one can characterise two disjoint trees on the heap that do not share any cells. Both trees occupy different portions of storage. □

2.2 Program Constructs for Resource Manipulation

We now introduce the program constructs associated with the original approach of separation logic [Rey02]. Like in the assertion part, additional program constructs are introduced for changing the dynamically allocated resources. Syntactically, the program commands are given by

$$\begin{aligned} \mathit{comm} ::= {} & \mathit{var} := \mathit{exp} \mid \mathsf{skip} \mid \mathit{comm}\,;\,\mathit{comm} \\ & \mid \mathsf{if}\ \mathit{bexp}\ \mathsf{then}\ \mathit{comm}\ \mathsf{else}\ \mathit{comm} \mid \mathsf{while}\ \mathit{bexp}\ \mathsf{do}\ \mathit{comm} \\ & \mid \mathsf{newvar}\ \mathit{var}\ \mathsf{in}\ \mathit{comm} \mid \mathsf{newvar}\ \mathit{var} := \mathit{exp}\ \mathsf{in}\ \mathit{comm} \\ & \mid \mathit{var} := \mathsf{cons}\,(\mathit{exp}, \ldots, \mathit{exp}) \\ & \mid \mathit{var} := [\mathit{exp}] \mid [\mathit{exp}] := \mathit{exp} \\ & \mid \mathsf{dispose}\ \mathit{exp}\,. \end{aligned}$$

We only give explanations for the heap changing commands since the other ones are well-known from the theory of Hoare logic. In particular, we provide a *small-step operational semantics* (see e.g., [Plo04]) given by a transition relation $\rightsquigarrow$ that describes the effects of a command on an arbitrary input state according to [Rey02]. The result of a computation either equals a state (s', h') or the execution aborts if it terminates. Notationally, we write in former case $\langle C, (s, h)\rangle \rightsquigarrow (s', h')$ and for the latter $\langle C, (s, h)\rangle \rightsquigarrow \mathsf{abort}$ where C is a program command formed according to the above syntax, (s, h) an initial and (s', h') a final state of the execution. A non-terminating command will lead to another configuration $\langle C', (s', h')\rangle$, where the command C' denotes a remaining execution and (s', h') is an intermediate state of the whole execution. The case of program abortion will appear for example when referencing non-allocated resources or assigning values to non-allocated heap cells. This treatment with a distinguished behaviour of faulting and non-terminating commands is needed to ensure validity of concepts that we introduce later.

Next, given a store s the command $v := \mathsf{cons}(e_1, ..., e_n)$ allocates n cells with e_i^s as the contents of the i-th cell. The cells form an unused contiguous region on the heap for which the starting address is chosen non-deterministically, hence it is unknown [YO02]. The address of the first cell is then stored in v while the rest of the cells can be addressed indirectly via the start address. Its operational semantics is defined by

$$\frac{a, \ldots, a + n - 1 \in \mathit{Addresses} - \mathit{dom}(h)}{\langle v := \mathsf{cons}(e_1, ..., e_n), (s, h)\rangle \rightsquigarrow ((v, a) \mid s, \{(a, e_1^s), \ldots, (a + n - 1, e_n^s)\} \mid h)} .$$

The premise $a, \ldots, a + n - 1 \in \mathit{Addresses} - \mathit{dom}(h)$ of that inference rule ensures that n unallocated addresses are available and can be allocated in h. By contrast to the commands in the following, abortion is not considered for allocation commands. The premise is ensured by the definition of an infinite set of available addresses in *Addresses* while heaps are defined to involve only a finite domain or set of allocated addresses. The reason for this definition is that the contiguous heap cells are chosen non-deterministically to obtain soundness of a central inference rule of separation logic that we introduce later.

We continue with commands of the form $v := [e]$ which are dereferencing assignments. The value e^s (corresponding to `*e` in the programming language `C`) needs to be a previously allocated address on the heap for a non-aborting execution of that command, i.e., $e^s \in \mathit{dom}(h)$ for an involved heap h. After its execution, the variable v on the store holds the contents of the dereferenced heap cell:

$$\frac{e^s \in \mathit{dom}(h)}{\langle v := [e], (s, h)\rangle \rightsquigarrow ((v, h(e^s)) \mid s, h)} , \qquad \frac{e^s \notin \mathit{dom}(h)}{\langle v := [e], (s, h)\rangle \rightsquigarrow \mathsf{abort}} .$$

Conversely, an execution of the command $[e_1] := e_2$ for *exp* - expressions e_1, e_2 assigns

the value of e_2 to the contents of the heap cell with address e_1:

$$\frac{e^s \in dom(h)}{\langle [e_1] := e_2, (s,h)\rangle \rightsquigarrow (s, (e_1^s, e_2^s) \mid h)}\,, \qquad \frac{e^s \notin dom(h)}{\langle [e_1] := e_2, (s,h)\rangle \rightsquigarrow \mathsf{abort}}\,.$$

Finally, the command $\mathsf{dispose}\ e$ is used for deallocating the heap cell at the address e^s. The disposed cell is not valid any more on the heap, i.e., dereferencing the value in e^s would cause a fault in the program execution. In particular, in the special case where e is a single program variable v, the address of the invalid heap cell remains stored there and hence is still saved in the store. The semantics is given by

$$\frac{e^s \in dom(h)}{\langle\, \mathsf{dispose}\, e, (s,h)\rangle \rightsquigarrow (s, h - (e^s, h(e^s)))}\,, \qquad \frac{e^s \notin dom(h)}{\langle\, \mathsf{dispose}\, e, (s,h)\rangle \rightsquigarrow \mathsf{abort}}\,.$$

With the semantics of the heap-manipulating program commands we now continue with the concept and semantics of Hoare triples in separation logic. By the inclusion of possibly aborting executions of commands, their semantics is slightly different from treatments of standard Hoare triples.

Definition 2.2.1 (Hoare triples in separation logic)
For commands C and assertions p, q the *Hoare triple* $\{p\}\, C\, \{q\}$ for *partial correctness* holds iff for all states $(s,h) \models p$ implies that

- $\neg(\langle C, (s,h)\rangle \rightsquigarrow^* \mathsf{abort})$,
- $\langle C, (s,h)\rangle \rightsquigarrow^* (s',h')$ implies $(s',h') \models q$,

where $\rightsquigarrow^*$ denotes the reflexive transitive closure of $\rightsquigarrow$.

Informally, a judgement $\{p\}\, C\, \{q\}$ is valid if any executions of the involved command C does not abort starting from any state that satisfies p. Moreover, if C terminates in a final state (s',h') then that state has to further satisfy q. A variant of this definition for the case of total correctness would additionally require that $(s,h) \models p$ also implies that any execution of $\langle C, (s,h)\rangle$ terminates. Examples for this definition of Hoare triples with can valid pre- and postconditions for heap-manipulating commands given in Figure 2.2.

In particular, almost all well-known and standard inference rules of Hoare logic are still valid. A small collection of important and frequently used rules are listed in Figure 2.3. Note, that all inference rules except the while - rule are valid in a partial and total correctness interpretation. For the latter case and while - loops it is required to add an additional termination argument to the premise of the corresponding rule.

$$\begin{array}{ccc} \{\exists v.\ e_1 \mapsto v\} & [e_1] := e_2 & \{e_1 \mapsto e_2\} \\ \{\,\mathsf{emp}\,\} & v := \mathsf{cons}(e_1, e_2) & \{v \mapsto e_1, e_2\} \\ \{e \mapsto v'\} & v := [e] & \{e \mapsto v' \wedge v = v'\} \\ \{e_1 \mapsto e_2\} & \mathsf{dispose}\,(e_1) & \{\,\mathsf{emp}\,\} \end{array}$$

Figure 2.2: Examples of Hoare triples in separation logic.

$$\frac{\{p_1\}\,C\,\{q_1\} \quad \{p_2\}\,C\,\{q_2\}}{\{p_1 \vee p_2\}\,C\,\{q_1 \vee q_2\}} \qquad \frac{\{p_1\}\,C\,\{q_1\} \quad \{p_2\}\,C\,\{q_2\}}{\{p_1 \wedge p_2\}\,C\,\{q_1 \wedge q_2\}} \qquad \frac{}{\{p[e/x]\}\,x := e\,\{p\}}$$

$$\frac{\{p\}\,C\,\{q\}}{\{\exists x.\,p\}\,C\,\{\exists x.\,q\}} \qquad \frac{\{p\}\,C\,\{q\}}{\{\forall x.\,p\}\,C\,\{\forall x.\,q\}} \qquad \frac{}{\{p\}\,\mathsf{skip}\,\{p\}}$$

$$\frac{p_1 \to p_2 \quad \{p_2\}\,C\,\{q_2\} \quad q_2 \to q_1}{\{p_1\}\,C\,\{q_1\}} \qquad \frac{\{p\}\,C_1\,\{r\} \quad \{r\}\,C_2\,\{q\}}{\{p\}\,C_1\,;\,C_2\,\{q\}}$$

$$\frac{\{p \wedge b\}\,C_1\,\{q\} \quad \{p \wedge \neg b\}\,C_2\,\{q\}}{\{p\}\ \mathsf{if}\ b\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\,\{q\}} \qquad \frac{\{p \wedge b\}\,C\,\{p\}}{\{p\}\ \mathsf{while}\ b\ \mathsf{do}\ C\,\{p \wedge \neg b\}}$$

Figure 2.3: Hoare logic inference rules.

An example of an inference rule that is valid in Hoare logic but false in separation logic according to [Rey09] is the following *rule of constancy* :

$$\frac{\{p\}\,C\,\{q\}}{\{p \wedge r\}\,C\,\{q \wedge r\}}\,,$$

where $\mathrm{FV}(r) \cap \mathrm{MV}(C) = \emptyset$. The side condition on the mentioned variables means that the command C is not allowed to modify any variable occurring free in the assertion r. A definition of $\mathrm{MV}(C)$ can be found in Appendix A.3. In a Hoare logic setting this is valid, since the assertions involved only make assumptions about store variables while in separation logic assertions of the form $p \wedge r$ can also make assumptions about the heap. In particular, the semantics of the logical conjunction is that p as well as r hold on the same heap and thus e.g., in the concrete instantiation $x \mapsto 3 \wedge y \mapsto 3$ the variable x and y would be aliases. By $\mathrm{MV}([x] := 4) = \emptyset$ and $y \mapsto 4 \wedge y \mapsto 3 \Leftrightarrow \mathsf{false}$ it is therefore not difficult to see that the following instantiation is invalid in separation logic

$$\frac{\{x \mapsto 3\}\,[x] := 4\,\{x \mapsto 4\}}{\{x \mapsto 3 \wedge y \mapsto 3\}\,[x] := 4\,\{x \mapsto 4 \wedge y \mapsto 3\}}\,.$$

To overcome this issue in separation logic O'Hearn and others replaced the Boolean conjunction in this rule with the separating conjunction $*$. This resulted in a powerful and central inference rule that enabled modular and local reasoning about parts of a program which can be further embedded into a larger context under a mild restriction on the set of involved variables. This basically reflects the power of separation logic and explains it impact on program verification. We will take a closer look at that inference rule in the next section.

2.3 The Frame Rule

The central tool of separation logic which makes that approach so popular and useful for concrete verification tasks is the so-called *frame rule* [ORY01]. It allows in combination with the separating conjunction, local reasoning about parts of the state that get changed by a corresponding program, and further enables the embedding of the resulting verification into a larger or more global context. For assertions p, q, r and command C it reads

$$\frac{\{p\}\, C\, \{q\}}{\{p * r\}\, C\, \{q * r\}}$$

assuming $\mathrm{FV}(r) \cap \mathrm{MV}(C) = \emptyset$ as in the case of the rule of constancy, i.e., all free variables of the assertion r are not modified by the command C. First, the premise of the rule ensures that any execution of C starting in a state satisfying p will end in a final state that satisfies q if it terminates. Now, the conclusion considers such executions in a consistent extension of the initial and final heaps with additional disjoint heap cells that satisfy r. As a concrete example consider the following instance of the frame rule using mutation commands:

$$\frac{\{x \mapsto v\}\, [x] := k\, \{x \mapsto k\}}{\{x \mapsto v * y \mapsto l\}\, [x] := k\, \{x \mapsto k * y \mapsto l\}}\,.$$

For the set of modified variables we have $\mathrm{MV}([x] := k) = \emptyset$. Hence the side condition is trivially satisfied. The precondition of the conclusion implicitly states that the addresses stored in x and y need to be different from each other. Therefore, the premise allows a local proof of the mutation command on the cell at address x without any effects on the additional cell at address y. The main idea is that any execution of C will not touch or modify any of the disjoint resources characterised by r since it is not required for a non-aborting execution. Hence a "local" proof of $\{p\}\, C\, \{q\}$ will extend to a "global" proof in the larger context extended by a frame r. A standard proof of that rule (see e.g. [YO02]) requires two further assumptions on the semantic foundation:

SAFETY MONOTONICITY: If a command C does not abort when starting an execution from a state (s, h), then C can also successfully run on states with a larger heap component, i.e., (s, h') with $h \subseteq h'$. This is formally expressed as

$$\neg(\langle C, (s, h)\rangle \rightsquigarrow^* \mathsf{abort}) \;\Rightarrow\; \neg(\langle C, (s, h')\rangle \rightsquigarrow^* \mathsf{abort}).$$

FRAME PROPERTY: Every execution of a command C can be tracked back to an execution of C running on states with possibly smaller heaps. By this, untouched allocated resources that do not affect any execution of C can be omitted. This reads formally for heaps h_0, h_1 with $dom(h_0) \cap dom(h_1) = \emptyset$ as

$$\begin{aligned}&(\neg(\langle C, (s, h_0)\rangle \rightsquigarrow^* \mathsf{abort}) \wedge \langle C, (s, h_0 \cup h_1)\rangle \rightsquigarrow^* (s', h')) \;\Rightarrow \\ &\qquad \exists\, h_0' : \langle C, (s, h_0)\rangle \rightsquigarrow^* (s', h_0') \wedge h' = h_0' \cup h_1 \wedge dom(h_0') \cap dom(h_1) = \emptyset.\end{aligned}$$

For establishing validity of the frame rule there also exists another approach [Vaf11] that uses a notion of *configuration safety* that inductively ensures non-aborting executions in terms of the operational semantics w.r.t. the steps a command can execute. By this, validity of Hoare triples is given if a command can safely execute all of its steps. We will consider for this thesis only the above defined properties and provide fully algebraic and pointfree characterisations of them that will consequently enable an abstract and generalised proof of the frame rule.

Chapter 3

Algebraic Spatial Assertions

An abstract and algebraic treatment of the assertion part of separation logic is presented, in particular of *separating conjunction*. For an adequate abstraction we start with an embedding of assertions into a set-based model that allows a treatment in a calculational style. In particular, we describe a translation of the spatial operators into that setting. This concrete model is further abstracted into the algebraic structure of so-called quantales in which assertions are represented as elements of the algebra. Using this algebra, different behaviours of special classes of assertions are characterised in a pointfree fashion by simple (in)equations. Moreover, this entails abstract and simple proofs of properties which can be checked and largely automated using general theorem proving systems. Another advantage of the algebraic view is that it yields new insights on separation logic by the application of the obtained results to a wide range of concrete models.

3.1 A Denotational Model for Assertions

A common methodology for providing a denotational model for logical assertions is given by an embedding of the satisfaction-based semantics for single states into an equivalent set-based and therefore pointfree setting. By this, every assertion will be associated with the set of all states that satisfy the corresponding assertion. We basically follow the approach of [DHM09, DHM10]. Concretely, for an arbitrary assertion p formed using the syntax given in Section 2.1 we define its set-based semantics as

$$[\![p]\!] =_{df} \{(s,h) : s,h \models p\} .$$

Clearly, by this definition all well-known Boolean connectives on assertions directly coincide with corresponding set-based operations where $|$ denotes the update operation on partial functions defined in Equation (2.1) and $\overline{\ }$ represents set complementation w.r.t. the carrier set *States*, i.e., for a set of states X we have $\overline{X} = \mathit{States} - X$. One inductively obtains the following equations for the standard logic connectives,

$$\begin{aligned}
[\![\neg p]\!] &= \{(s,h) : s,h \not\models p\} = \overline{[\![p]\!]}\,, \\
[\![p \vee q]\!] &= [\![p]\!] \cup [\![q]\!]\,, \\
[\![p \wedge q]\!] &= [\![p]\!] \cap [\![q]\!]\,, \qquad [\![p \to q]\!] = \overline{[\![p]\!]} \cup [\![q]\!]\,, \\
[\![\forall v : p]\!] &= \{(s,h) : \forall x \in \mathbb{Z} : (v,x) \,|\, s,h \models p\} \\
&= \bigcap_{x \in \mathbb{Z}} \{(s,h) : ((v,x) \,|\, s,h) \in [\![p]\!]\}\,, \\
[\![\exists v : p]\!] &= \overline{[\![\forall v : \neg p]\!]} = \{(s,h) : \exists x \in \mathbb{Z}.\ (v,x) \,|\, s,h \models p\} \\
&= \bigcup_{x \in \mathbb{Z}} \{(s,h) : ((v,x) \,|\, s,h) \in [\![p]\!]\}\,.
\end{aligned}$$

As particular cases, $[\![\mathsf{true}]\!] = \mathit{States}$ and $[\![\mathsf{false}]\!] = \emptyset$. Similarly, set-based variants for the assertion emp that characterises the empty heap and $e_1 \mapsto e_2$ that denotes a single cell heap can be given by

$$[\![\mathsf{emp}]\!] = \{(s,h) : h = \emptyset\} \qquad \text{and} \qquad [\![e_1 \mapsto e_2]\!] = \left\{(s,h) : h = \left\{\left(e_1^s, e_2^s\right)\right\}\right\}.$$

For an adequate reformulation of separating conjunction $*$ on sets of states expressing heap disjointness we obtain

$$[\![p * q]\!] = [\![p]\!] \cup [\![q]\!]\,,$$

where for sets $P, Q \in \mathcal{P}(\mathit{States})$ we define

$$P \cup Q =_{df} \{(s, h \cup h') : (s,h) \in P,\ (s,h') \in Q,\ \mathit{dom}(h) \cap \mathit{dom}(h') = \emptyset\}\,.$$

Assuming that the considered states involve the same stores and address-disjoint heaps, this operator exactly renders the semantics of separating conjunction in a set-based fashion. Generally, this construction yields an algebraic embedding of separation logic assertions into an abstract calculus by viewing the constructed sets of states as elements of a specific structure that we discuss in the following. A central requirement for this task involves the inclusion of algebraic counterparts of all the above set-based operations. Especially due to the usage of possible infinite intersections and unions it turned out that an appropriate algebraic structure for this purpose are *quantales* which have been introduced in [Mul86, Ros90].

Definition 3.1.1 (Quantale)

(a) A *quantale* is a structure $(S, \leq, \cdot, 1)$ such that

- $(S, \leq)$ is a complete lattice where, for $T \subseteq S$, the element $\bigsqcup T$ denotes the supremum of T and $\bigsqcap T$ its infimum,
- $(S, \cdot, 1)$ is a monoid,
- multiplication distributes over arbitrary suprema, i.e., for $a \in S$ and $T \subseteq S$,

$$a \cdot (\bigsqcup T) = \bigsqcup\{a \cdot b : b \in T\} \quad \text{and} \quad (\bigsqcup T) \cdot a = \bigsqcup\{b \cdot a : b \in T\}. \tag{3.1}$$

The least and greatest element of a quantale w.r.t. $\leq$ are denoted by 0 and $\top$, resp. Binary infima and suprema of two elements $a, b \in S$ are denoted by $a \sqcap b$ and $a + b$, resp. We assume that $\sqcap$ binds tighter than $+$.

(b) A quantale is called *commutative* iff $a \cdot b = b \cdot a$ for all $a, b \in S$.

(c) A quantale is called *Boolean* iff its underlying lattice is distributive, i.e., for all $a, b, c \in S$

$$a \sqcap (b + c) = a \sqcap b + a \sqcap c \quad \text{and} \quad a + b \sqcap c = (a + b) \sqcap (a + c),$$

and is complemented. Complementation will be denoted by $\overline{}$. Moreover, the greatest element $\top$ is defined by $\overline{0}$.

Note that by this definition $+$ and $\sqcap$ are commutative, associative and idempotent. The former operator has the unit 0 and annihilator $\top$ while conversely $\sqcap$ has $\top$ as its unit and 0 as its annihilator. The natural order $\leq$ satisfies $a \leq b \Leftrightarrow a + b = b \Leftrightarrow a \sqcap b = a$ for arbitrary elements $a, b \in S$.

Furthermore in a quantale one can derive that the least element satisfies $\bigsqcup \emptyset = 0$. Due to Equation (3.1) this immediately implies that $\cdot$ is strict in both arguments, i.e., we have $0 \cdot a = 0 = a \cdot 0$ for all $a \in S$ and hence 0 is an annihilator.

The following equivalences are valid in quantales and will facilitate inequational reasoning in proofs provided in later sections:

$$a + b \leq c \Leftrightarrow a \leq c \wedge b \leq c \quad \text{and} \quad a \leq b \sqcap c \Leftrightarrow a \leq b \wedge a \leq c. \tag{3.2}$$

In the case of Boolean quantales we have

$$a \sqcap b \leq c \Leftrightarrow a \leq b \rightarrow c. \tag{shu}$$

where $b \rightarrow c =_{df} \overline{b} + c$. This property is called *shunting* and entails in particular, $a \sqcap b \leq 0 \Leftrightarrow b \leq \overline{a}$.

For easier readability we suppose that multiplication $\cdot$ binds tighter than $\sqcap$ and $+$.

The assumption that $(S, \leq)$ is a complete lattice guarantees the existence of infinite suprema and infima as required for a complete algebraic treatment of separation logic assertions. We can now conclude the following result.

Theorem 3.1.2 *The structure* $\mathsf{AS} =_{df} (\mathcal{P}(States), \subseteq, \cup, [\![\,\mathsf{emp}\,]\!])$ *is a commutative and Boolean quantale with* $P + Q = P \cup Q$.

Proof. First, it is not difficult to see that $(\mathcal{P}(States), \subseteq)$ forms a complete and distributive lattice where $\bigsqcup$ and $\bigsqcap$ coincide with $\bigcup$ and $\bigcap$, respectively. Moreover, that $\cup$ is associative, commutative and has $[\![\,\mathsf{emp}\,]\!]$ as its unit follows from the definitions and pointwise lifting. Hence, $(\mathcal{P}(States), \cup, [\![\,\mathsf{emp}\,]\!])$ represents a commutative monoid. The distributivity laws of separating conjunction can also be lifted to the set-based setting and extended in AS to arbitrary unions. Finally, Boolean complements in AS can be obtained using set-complementation $\overline{}$. □

By Theorem 3.1.2 it is obvious that $[\![\,\mathsf{true}\,]\!]$ coincides with $\top$ and $[\![\,\mathsf{false}\,]\!]$ with 0. Binary intersections and unions are abstracted to $\sqcap$ and $+$, respectively. As already mentioned a concrete instance of the infinite distributivity laws in Equation (3.1) can be found in logical formulas involving existential quantifications like $p * (\exists v.\ q) \Leftrightarrow \exists v.\ p * q$ and its symmetric variant for arbitrary assertions p, q and variable $v \notin \mathrm{FV}(p)$.

In the case of arbitrary infima, only validity of inequational variants of distributivity laws can be obtained, i.e., for an abstract quantale $(S, \leq, \cdot, 1)$ and subset $T \subseteq S$ we have

$$(\bigsqcap T) \cdot b \leq \bigsqcap \{a \cdot b : a \in T\} \quad \text{and} \quad a \cdot (\bigsqcap T) \leq \bigsqcap \{a \cdot b : b \in T\}. \tag{3.3}$$

Special cases or instances of these abstract laws w.r.t. AS are given in separation logic e.g., by

$$(p \wedge q) * r \;\Rightarrow\; (p * r) \wedge (q * r) \quad \text{and} \quad (\forall x.\ p) * q \;\Rightarrow\; \forall x.\ p * q\,.$$

A further useful property of quantales is that by the above inequational distributivity laws, multiplication is isotone in both arguments, i.e., for elements a, b, c, d: $a \leq b \wedge c \leq d \Rightarrow a \cdot b \leq c \cdot d$. This can be translated into separation logic for adequate assertions p, q, r, s to the valid inference rule

$$\frac{p \;\Rightarrow\; r \quad q \;\Rightarrow\; s}{p * q \;\Rightarrow\; r * s}\,.$$

More laws and examples can be found in [Dan09]. Next, we derive an algebraic characterisation for the remaining separation logic operations. First, we start with a

treatment of the separating implication. Its logic-based application is given in [Rey02, Rey09] by instantiations of so-called *currying* and *decurrying* inference rules. For arbitrary assertions p, q, r separating implication and separating conjunction satisfy the following interplay:

$$\frac{p * q \;\Rightarrow\; r}{p \;\Rightarrow\; (q \mathrel{-\!\!*} r)} \quad \text{(currying)} \qquad\qquad \frac{p \;\Rightarrow\; (q \mathrel{-\!\!*} r)}{p * q \;\Rightarrow\; r} \quad \text{(decurrying)}\,.$$

From an algebraic viewpoint these laws are very similar to the Galois equivalences characterising residuals in quantales [Bir67, Lam68]. Such elements represent the greatest solutions x w.r.t. the order $\leq$ of the inequation $a \cdot x \leq b$ for arbitrary elements a, b of a quantale.

Definition 3.1.3 (Residuals)

(a) In any quantale, the *right residual* $a\backslash b$ exists and is characterised by the Galois connection

$$x \leq a\backslash b \;\Leftrightarrow_{df}\; a \cdot x \leq b\,. \tag{3.4}$$

$a\backslash b$ as the greatest solution of the inequation $a \cdot x \leq b$ denotes a pseudo-inverse to multiplication.

(b) Symmetrically, the *left residual* b/a can be defined by

$$x \leq b/a \;\Leftrightarrow_{df}\; x \cdot a \leq b\,. \tag{3.5}$$

In the case of a commutative quantale both residuals coincide, i.e., $a\backslash b = b/a$.

Note that in quantales residuals do always exist by the assumption of a complete underlying lattice that guarantees the existence of arbitrary suprema. In the concrete quantale AS, we will provide a proof that algebraic residuals coincide conceptually in the set-based setting with separating implication which reads

$$\begin{aligned}[\![p \mathrel{-\!\!*} q]\!] = \{(s,h) :\; &\forall h' \in \mathit{Heaps} : (\, \mathit{dom}(h) \cap \mathit{dom}(h') = \emptyset \wedge (s,h') \in [\![p]\!]) \\ &\Rightarrow (s, h \cup h') \in [\![q]\!]\}\,.\end{aligned}$$

Residuals have been researched for already several decades and hence a large amount of general results can be immediately applied to the concrete quantale AS and thus become consequences in separation logic. For notational benefit we use in the following the same symbols for residuals in AS and abstract quantales.

Lemma 3.1.4 *In* AS, $[\![p \mathrel{-\!\!*} q]\!] = [\![p]\!]\backslash[\![q]\!] = [\![q]\!]/[\![p]\!]$.

Proof. By set theory and definition of $\uplus$, we have

$$\begin{array}{ll} & (s,h) \in [\![p \mathrel{-\!\!*} q]\!] \\ \Leftrightarrow & \forall\, h' : (dom(h) \cap dom(h') = \emptyset \wedge (s,h') \in [\![p]\!] \Rightarrow (s, h \cup h') \in [\![q]\!]) \\ \Leftrightarrow & \{(s, h \cup h') : dom(h) \cap dom(h') = \emptyset \wedge (s,h') \in [\![p]\!]\} \subseteq [\![q]\!] \\ \Leftrightarrow & \{(s,h)\} \uplus [\![p]\!] \subseteq [\![q]\!] \end{array}$$

and therefore, for arbitrary set R of states,

$$\begin{array}{ll} & R \subseteq [\![p \mathrel{-\!\!*} q]\!] \\ \Leftrightarrow & \forall\, (s,h) \in R : (s,h) \in [\![p \mathrel{-\!\!*} q]\!] \\ \Leftrightarrow & \forall\, (s,h) \in R : \{(s,h)\} \uplus [\![p]\!] \subseteq [\![q]\!] \\ \Leftrightarrow & R \uplus [\![p]\!] \subseteq [\![q]\!] \,. \end{array}$$

Hence, by definition of right residuals, $[\![p \mathrel{-\!\!*} q]\!] = [\![p]\!] \backslash [\![q]\!]$. The second equation follows immediately since $\uplus$ in AS commutes (cf. Theorem 3.1.2). □

A similar result was stated in [IO01]. There it was mentioned that the assertional part of separation logic is an instance of an abstract approach called the *logic of bunched implications* [OP99]. We will elaborate on this in Section 3.1.1.

In our setting, by Lemma 3.1.4 the currying and decurrying inference rules become theorems in the assertion quantale AS, and hence also all well-known laws about $\mathrel{-\!\!*}$ are now theorems of the standard theory of residuals (e.g. [BJ72]). As an example a frequently used inference rule in separation logic is

$$\frac{}{q \ast (q \mathrel{-\!\!*} p) \Rightarrow p}$$

which describes the general behaviour of separating implication that whenever a heap satisfying q gets combined with a disjoint for which $q \mathrel{-\!\!*} p$ holds then the whole heap will satisfy p. Abstractly, we can show, as a direct consequence of the definition of residuals, that for arbitrary elements $a, b \in S$ the inequality $b \cdot (b \backslash a) \leq a$ holds.

Lemma 3.1.5 *$b \cdot (b\backslash a) \leq a$ and symmetrically $(a/b) \cdot b \leq a$ is valid.*

Proof. By Definition 3.1.3 we infer $b \cdot (b\backslash a) \leq a \Leftrightarrow b\backslash a \leq b\backslash a \Leftrightarrow \mathsf{true}$. □

Now, setting $a = [\![p]\!]$ and $b = [\![q]\!]$ one obtains validity of the inequation in AS and hence also soundness of the above inference rule in separation logic.

For later calculational proofs we list a couple of helpful properties. Right residuals are anti-disjunctive in their first argument and conjunctive in their second one, i.e., for a set $T \subseteq S$ of quantale S

$$y\backslash(\bigsqcap T) = \bigsqcap \{y\backslash x : x \in T\} \quad \text{and} \quad (\bigsqcup T)\backslash x = \bigsqcap \{y\backslash x : y \in T\} \,.$$

In the semantics of separation logic this entails as an example validity of the logical equivalence $p \mathrel{-\!\!*} (\forall\, v.\ q) \;\Leftrightarrow\; \forall\, v.\ p \mathrel{-\!\!*} q$ where $v \notin \mathrm{FV}(p)$. Abstractly, the above laws immediately imply for arbitrary elements z the following consequence

$$x \leq y \;\;\Rightarrow\;\; z\backslash x \leq z\backslash y \;\;\wedge\;\; y\backslash z \leq x\backslash z\,.$$

Another law involves e.g., a further characterisation of $\top$ by $y\backslash\top = \top = 0\backslash x$. Many of these laws are proved algebraically in [Dan09] and have also been automated.

As a last ingredient and for completeness reasons we include into AS another operator that is closely related to separating implication. It is called *septraction* in the separation logic literature e.g. in [VP07] and is defined as follows.

Definition 3.1.6 (Septraction)
For assertions p, q septraction is defined by $p \mathrel{-\!\!\circledast} q \;\Leftrightarrow_{df}\; \neg(\, p \mathrel{-\!\!*} (\neg q))$.

The intuition of this operator can be given by the following lemma. We provide its pointwise meaning using the carrier set *States*.

Lemma 3.1.7 $s, h \models p \mathrel{-\!\!\circledast} q \;\;\Leftrightarrow\;\; \exists\, \hat{h} : h \subseteq \hat{h} \wedge s, \hat{h} - h \models p \wedge s, \hat{h} \models q\,.$

The proof of Lemma 3.1.7 is deferred to Appendix A. Informally, if a heap h satisfies $p \mathrel{-\!\!\circledast} q$, then it can be extended with a heap that satisfies p so that q holds for the resulting one. Equivalently, one can also quantify over the existence of a remaining disjoint heap satisfying p. In contrast to separating implication it comes with angelic behaviour since it only quantifies existentially over the remaining heap portion where p holds while in the case of the implicational version its condition needs to be fulfilled by all heaps. Due to this one often refers to septraction as the existential separating implication.

Using the above definitions we can analogously derive a set-based version of septraction in AS. Interestingly, its logical pointfree definition directly coincides in AS with so-called detachment operators [Bir67] that also do exist in arbitrary Boolean quantales. As in the case of residuals we notationally also use the same symbol for detachments in AS and arbitrary Boolean quantales.

Definition 3.1.8 (Detachments)
In a Boolean quantale, the *left detachment* can be defined based on the left residual for elements a, b by

$$a \rfloor b \;=_{df}\; \overline{a\backslash \overline{b}}\,.$$

Symmetrically the *right detachment* is defined by $a \lfloor b \;=_{df}\; \overline{a/b}$. If the underlying quantale is commutative $a \rfloor b = b \lfloor a$.

Therefore, in the quantale AS one obtains by Theorem 3.1.2

$$[\![p \mathrel{-\!\!\circledast} q]\!] = [\![\neg(p \mathrel{-\!\!*} (\neg q))]\!] = \overline{[\![p]\!] \backslash \overline{[\![q]\!]}} = [\![p]\!] \rfloor [\![q]\!] .$$

As in the case of residuals and separating implication, a large amount of already known laws for detachments immediately applies to the assertion quantale AS and thus to the assertional part of separation logic.

Detachments are isotone in both arguments. Moreover, from the characterising Galois connection of residuals one obtains by de Morgan's laws the exchange properties

$$a \lfloor b \leq x \Leftrightarrow \overline{x} \cdot b \leq \overline{a} \quad \text{and} \quad a \rfloor b \leq x \Leftrightarrow a \cdot \overline{x} \leq \overline{b} . \tag{exc}$$

Another important consequence are the *Dedekind rules* [JT51]

$$a \sqcap b \cdot c \leq (a \lfloor c \sqcap b) \cdot c \quad \text{and} \quad a \sqcap b \cdot c \leq b \cdot (b \rfloor a \sqcap c) . \tag{Ded}$$

In separation logic these inequations would translate for adequate assertions p, q, r to validity of the implication $p \wedge q * r \Rightarrow q * ((q \mathrel{-\!\!\circledast} p) \wedge r)$. Concretely, it asserts that if some heap h that satisfies p can be split into disjoint portions for which q and r hold then h can also be split into one part satisfying q and a remaining disjoint one for which r and $q \mathrel{-\!\!\circledast} p$ hold. In the following section we will provide more properties for septraction and separating implication interacting with assertions that come with special behaviour. In particular, this will demonstrate the simplicity and benefits of our approach for deriving several frequently required theorems in a pointfree style. Finally, to round off our derivations on the algebraic structure we provide in Table 3.1 a notational overview of the concrete powerset structure modelling spatial assertions and the correspondences between the operations of separation logic and the abstract algebra.

Name in Logic	**SL**	**AS**	**Quantales**
disjunction	$\vee$	$\cup$	$+$
conjunction	$\wedge$	$\cap$	$\sqcap$
negation	$\neg$	$\overline{}$	$\overline{}$
implication	$\Rightarrow$	$\subseteq$	$\leq$
separating conjunction	$*$	$\dot{\cup}$	$\cdot$
separating implication	$\mathrel{-\!\!*}$	$\backslash$	$\backslash$
septraction	$\mathrel{-\!\!\circledast}$	$\rfloor$	$\rfloor$

Figure 3.1: Notations of operators in separation logic, AS and abstract quantales.

3.1.1 Related Work: BI Algebras

Abstract and algebraic structures for the spatial assertions of separation logic have also been investigated in earlier approaches. Originally, in 1999 O'Hearn and Pym developed a logical approach called the *logic of bunched implications* (BI) [OP99]. It introduced the general ideas of the structure of today's separation-logical assertions. The standard interpretations depending on the carrier set *States* was considered as an model of a Boolean variant of BI [IO01]. In contrast to classical logical approaches, BI comes with two different conjunction and implication operations, i.e., concretely they coincide in separation logic with $\wedge, *$ and $\rightarrow, \mathrel{-\!\!*}$. The spatial operations $*, \mathrel{-\!\!*}$ are also called multiplicative connectives while the remaining ones are named additive in the literature. Algebraic presentations of BI use as a starting base the structure of a Heyting algebra $(S, \leq)$, i.e., a lattice containing a greatest and least element w.r.t. $\leq$ and binary meets denoted by $a \sqcap b$ that are residuated. They represent a lower adjoint and have a corresponding upper adjoint $\rightarrow$ that is characterised by the Galois connection

$$a \sqcap b \leq c \Leftrightarrow a \leq b \rightarrow c\,.$$

Note that in any Boolean algebra this condition is always satisfied and stated in the present quantale-based approach as shunting (cf. (shu)). More interestingly, in addition to the assumed Heyting algebra one requires a further residuated commutative monoid structure denoted by $(S, *, \mathsf{emp})$ that similarly to the above satisfies

$$a * b \leq c \Leftrightarrow a \leq b \mathrel{-\!\!*} c\,.$$

Its purpose is to abstractly model the substructural part of separation logic given by separating conjunction. Concretely $*$ does not satisfy the weakening and contraction rules

$$a * b \leq a \quad \text{and} \quad a \leq a * a \tag{3.6}$$

in contrast to $\sqcap$, since otherwise both operations would coincide (cf. [OP99]). In sum, the full algebraic structure is called a *BI algebra*. Since separation logic in its early developments was provided as an intuitionistic logic [Rey00] and Heyting algebras model propositional versions of such logics, an adequate abstraction [Pym02, POY04] is found by these algebras. An approach to propositional versions of classical logics, e.g., separation logic in its nowadays version [Rey02] requires the extension of BI to a Boolean algebra by replacing the underlying Heyting algebra by a Boolean one. That approach is called a *Boolean BI algebra* and it differs from the algebraic treatment based on commutative Boolean quantales in not requiring the following structural assumptions:

- an underlying complete lattice involving infinite meets and joins,

- the associated infinite (semi)distributivity laws w.r.t. $*$.

Due to the characterisation of $*$ as a lower adjoint in the above Galois connection the second property would be immediately implied if arbitrary (infinite) meets and joins are available (e.g. [EKMS92, Möl99b]). Further considerations of Boolean BI algebras extended to complete lattices can be found in [BBTS05, BBTS07]. In particular, similar powerset constructions as presented in Section 3.1 are discussed and provided in those papers within a categorical setting called *BI hyperdoctrines*. More category theory related approaches involving infinite distributivity laws are discussed in [Pym02, POY04, Bie04] where especially topological representations are considered that also involve complete BI algebras.

The presented approach will stay within an algebraic treatment. Its main focus comprises, in addition to its abstract character, the further possibility of calculating a large set of theorems in fully pointfree way especially in combination with algebraic characterisations of special behaviour of certain assertion classes [Rey09] which are presented in the subsequent section. Due to the simple representations and largely first-order logic formulated (in)equations that setting also comes with the advantage to easily generate mechanised and (semi)automated proofs. This in turn guarantees more safety and confidence in the correctness of the presented derivations.

Finally, we also remark that, based on the logic of BI, abstract resource semantics and interpretations in a Kripke-style have been already derived in [Pym02]. These model-theoretic considerations entailed a pointwise definition of the separating implication that corresponds in a particular case to the definition based on the carrier set *States* [IO01]. Hence Lemma 3.1.4 expresses exactly the expected equalities by interpreting separating implication as residuals within Boolean quantales.

3.2 Characterising Behaviour Abstractly

With the developed foundation of the previous section we can now move one step further and consider the algebraic approach for characterising different classes of assertions in separation logic as presented in [Rey02, Rey09]. Motivated by the fact that certain consequences of such classes can be described in a pointfree fashion, the question arises whether it is possible to obtain a completely algebraic treatment of the assertion classes. As an example so-called *pure* assertions characterise on states (s, h) only conditions that do not involve the heap component, i.e., exclusively properties about store variables are expressed. This implies e.g., that logical and separating conjunction coincide for pure assertions P, Q, i.e.,

$$P * Q \Leftrightarrow P \wedge Q\,.$$

By results of the previous section it is an easy task to abstract this law to arbitrary quantales. In what follows, investigations are presented for the most common assertion classes to find suitable algebraic abstractions that enable the derivation of properties at the propositional level of separation logic as above. This immediately facilitates assertional reasoning, especially in the case of purely first-order logical characterisations that enable automation or at least mechanised proofs.

3.2.1 Intuitionistic Assertions

We start by considering the class of *intuitionistic* assertions. They reflect the intuitionistic behaviour of assertions in the early developments of [Rey00]. In more abstract settings as in the logic of BI, the intuitionistic behaviour is called *Kripke monotonicity* and assumed for Kripke interpretations that abstract heaps to the semantics of possible worlds [POY04]. In the concrete setting of *States* the behaviour is given in [IO01] as a *monotonicity condition* and can be described in the sense that intuitionistic assertions do not characterise the domain of a heap or the set of allocated heap cells exactly. Hence, an imprecision is introduced due to some additional set of anonymous cells that may reside on the heap. This can happen in the case when e.g., pointer references to some prior allocated storage are lost.

Following [Rey09], an assertion p is called *intuitionistic* iff

$$\forall\, s \in \mathit{Stores}, \forall\, h, h' \in \mathit{Heaps} : (\, h \subseteq h' \wedge s, h \models p\,) \;\Rightarrow\; s, h' \models p\,. \tag{3.7}$$

Intuitively, if a heap h that satisfies an intuitionistic assertion p then any larger heap, i.e., extended by arbitrary cells, still satisfies p. It turned out that this particular closure condition can be characterised within the quantale AS in a pointfree style.

Theorem 3.2.1 *In* AS *an element* $[\![p]\!]$ *is intuitionistic iff it satisfies*

$$[\![p]\!] \cup [\![\mathsf{true}]\!] \subseteq [\![p]\!]\,.$$

Proof. By definition of true, set theory, using for $(\Rightarrow)$ $h'' = h' - h$ and for $(\Leftarrow)$ that $\mathit{dom}(h) \cap \mathit{dom}(h'') = \emptyset \wedge h' = h \cup h'' \Rightarrow h'' = h' - h$, a logic step, definition of $*$,

$$\begin{array}{ll}
 & \forall\, s, h, h' :\ (\, h \subseteq h' \wedge s, h \models p) \;\Rightarrow\; s, h' \models p \\
\Leftrightarrow & \forall\, s, h, h' :\ (\, h \subseteq h' \wedge s, h \models p \wedge s, (h' - h) \models \mathsf{true}\,) \;\Rightarrow\; s, h' \models p \\
\Leftrightarrow & \forall\, s, h, h' :\ (\, s, h \models p \wedge s, (h' - h) \models \mathsf{true} \wedge \mathit{dom}(h) \cap \mathit{dom}(h' - h) = \emptyset \\
 & \qquad \wedge\ h' = h \cup (h' - h)\,) \;\Rightarrow\; s, h' \models p \\
\Leftrightarrow & \forall\, s, h, h' :\ (\, \exists\, h'' :\ s, h \models p \wedge s, h'' \models \mathsf{true} \wedge \mathit{dom}(h) \cap \mathit{dom}(h'') = \emptyset \\
 & \qquad \wedge h' = h \cup h'\,) \;\Rightarrow\; s, h' \models p
\end{array}$$

$$\begin{aligned}
&\Leftrightarrow \forall\, s,h' :\ (\exists\, h,h'' :\ s,h \models p \ \wedge\ s,h'' \models \mathsf{true} \ \wedge\ dom(h) \cap dom(h'') = \emptyset \\
&\qquad\qquad \wedge\, h \cup h'' = h'\,) \ \Rightarrow\ s,h' \models p \\
&\Leftrightarrow \forall\, s,h' :\ s,h' \models p * \mathsf{true} \ \Rightarrow\ s,h' \models p\,.
\end{aligned}$$

Now, the claim follows from the translation of assertions into AS given in Section 3.1. □

Thus, Theorem 3.2.1 yields the following definition by the abstraction of its result to arbitrary Boolean quantales.

Definition 3.2.2
In an arbitrary Boolean quantale S an element a is called *intuitionistic* iff it satisfies

$$a \cdot \top \leq a\,.$$

Note that this inequation can be strengthened to an equation since its converse holds for arbitrary Boolean quantales by neutrality of 1 and the fact that multiplication is isotone. Hence, $\top$ is characterised as a neutral element w.r.t. multiplication on the set of intuitionistic elements. Clearly, $\top$ is intuitionistic. In the case of separation logic, $\top$ coincides with true as a unit for separating conjunction $*$. This result is stated as a consequence e.g. in [IO01]. Abstractly, elements of the form $a \cdot \top$ are also called in the literature vectors or ideals. Those elements are well known, and therefore one can easily transfer many properties (e.g. [SS93, Mad06]) to this particular application domain without any additional effort. We list some of them to show again the advantages of the algebraic approach and start by some intuitive closure properties which can also be found in [Rey02].

Lemma 3.2.3 *Consider a commutative Boolean quantale S, intuitionistic elements $a, a_i \in S$ with $i \in \mathbb{N}$ and an arbitrary element $b \in S$. Then the following composed elements are also intuitionistic:*

(a) $a \cdot b$, *hence also* $a \cdot \top$,

(b) $b\backslash a$, *hence also* $\top\backslash a$,

(c) $\bigsqcap\{a_i : i \in \mathbb{N}\}$,

(d) $\bigsqcup\{a_i : i \in \mathbb{N}\}$.

Proof. The proof of part (a) is immediate from commutativity of $\cdot$ and the assumption. For part (b) we have $(b\backslash a) \cdot \top \leq b\backslash a \Leftrightarrow b \cdot (b\backslash a) \cdot \top \leq a$ by Definition 3.1.3 and the claim follows from $b \cdot (b\backslash a) \leq a$ by Lemma 3.1.5 and the assumption that a is intuitionistic. The remaining parts follow from Equation (3.3), Equation (3.1) and the assumptions. □

Note, that weaker assumptions in Lemma 3.2.3 can be used than e.g. provided in [Rey02]. We continue by particular laws that describe the interaction of $\cdot$ and $\sqcap$ involving intuitionistic assertions.

Lemma 3.2.4 *Consider a commutative Boolean quantale S, intuitionistic elements $a, a' \in S$ and arbitrary elements $b, c \in S$. Then*

(a) $(a \sqcap b) \cdot c \leq a \sqcap b \cdot c$,

(b) $(a \sqcap b) \cdot \top \leq a$,

(c) $a \cdot b \leq a \sqcap b \cdot \top$,

(d) $a \cdot a' \leq a \sqcap a'$.

Proof. To show (a) we calculate $(a \sqcap b) \cdot c \leq a \cdot c \sqcap b \cdot c \leq a \cdot \top \sqcap b \cdot c \leq a \sqcap b \cdot c$. Setting $c = \top$ in (a) the inequation of (b) follows from isotony of $\sqcap$. For a proof of (c) we know $a \cdot b \leq a \cdot \top \leq a$ and $a \cdot b \leq \top \cdot b = b \cdot \top$ by isotony and commutativity of $\cdot$. Again, Equation (3.2) shows the claim. Trivially (d) follows from (c) setting $b = a'$, applying the definition of intuitionistic elements and using isotony of $\sqcap$. □

Note that part (b) implies by isotony of multiplication the weakening rule for $*$ of BI algebras (cf. Equation (3.6)), i.e., $a * b \leq a$ assuming a is intuitionistic. Moreover, it is not difficult to see that the reverse inequations of the above given laws can not be obtained for the quantale AS considering translations of adequate separation logical assertions. In particular, part (d) above only validates an inequation on intuitionistic elements characterising the interplay of multiplication and meet, or concretely separation and logical conjunction, respectively. A counterexample for the opposite direction in that case is easily constructed with $a = a' = [\![x \mapsto 1 * \mathsf{true}]\!] = [\![x \mapsto 1]\!] \cup [\![\mathsf{true}]\!]$. Obviously a and a' are both intuitionistic. Hence the definitions (cf. Section 2.1) immediately imply that $a \cap a' = a \neq \emptyset$ and $a \cup a' = [\![x \mapsto 1]\!] \cup [\![\mathsf{true}]\!] \cup [\![x \mapsto 1]\!] \cup [\![\mathsf{true}]\!] = \emptyset$ since $\cup$ is commutative and $[\![x \mapsto 1]\!] \cup [\![x \mapsto 1]\!] = \emptyset$.

This further implies that even for intuitionistic elements, the contraction law $a \leq a \cdot a$ does not hold. We show in the subsequent section that a strengthening of the interplay given in part (d) to an equation can be guaranteed for assertions whose validity is independent of the underlying heap component within arbitrary states.

Finally, we remark that similar abstract characterisations as given in Definition 3.2.2 can also be found in [BBTS05, BBTS07] within a higher-order and also abstract setting of separation logic, called BI-hyperdoctrines. Intuitionistic assertions q can be found there under the definition of *monotone assertions* w.r.t. heaps h of states $(s, h) \in \mathit{States}$. A pointfree characterisation is given by a quantification over arbitrary assertions p in $\forall p.\ p * q \to q$. This closely correlates to our compact characterisation by interpreting p, q as elements of a commutative Boolean quantale and $\to$ as its associated natural order $\leq$. Then one can easily conclude

$$(\forall p.\ p \cdot q \leq q) \Leftrightarrow q \cdot \top \leq q,$$

where $\Rightarrow$ follows by commutativity of $\cdot$ and setting $p = \top$ and the reverse direction $\Leftarrow$ directly holds by isotony of multiplication.

3.2.2 Resource Independence

In this section special attention is paid to a particular class of assertions for which the behaviour can simply be described by not making any assumptions about the heap component of states, i.e., assertions that are valid for arbitrary heaps. Concrete instances within separation logic are given by any *bexp* - expression (cf. Section 2.1) like false, true or $e = e'$ which coincide with assertions of usual Hoare logic approaches [Hoa69]. Originally, in the separation logic literature such assertions appeared in [IO01, Rey02] under the notion of *pure assertions* . They are syntactically described there as assertions that do not contain emp and $\mapsto$ (w.r.t. the syntax given in Section 2.1). An exclusion of the spatial operations $*$ and $-\!\!*$ making assumptions about the underlying heaps in formulas is not required since these operations will collapse with logical conjunction and implication, respectively. A complete algebraic proof of this fact will be provided later.

Another application of pure assertions can be found in [PS11, PS12]. The setting in those papers considers an extended carrier set where each state is equipped with a map for modelling permission to access certain fields of objects. Separating conjunction is defined there to split permission maps while only allowing states with equal stores and heaps. An abstraction of pure assertions to an algebraic setting will also incorporate such particular models.

For establishing an algebraic characterisation of pure assertions we turn back to the standard setting based on the carrier set *States* . First we provide a common definition of pure assertions given by the following formula

$$p \text{ is } \textit{pure} \Leftrightarrow_{df} \ (\forall\, s \in \textit{States} : \forall\, h, h' \in \textit{Heaps} : s, h \models p \Leftrightarrow s, h' \models p\,) . \tag{3.8}$$

First, we immediately infer from the above definition that if p holds for any state at all then especially $s, \emptyset \models p$ holds. Trivially, then p is also satisfied by any heap larger than the empty one. Both facts will be used to derive a pointfree characterisation of pure assertions. Following the ideas introduced in [DHM09, DHM11] we can characterise pure assertions in the quantale AS follows.

Theorem 3.2.5 *In* AS *an element* $[\![p]\!]$ *is pure iff it satisfies*

$$[\![p]\!] = ([\![p]\!] \cap [\![\mathsf{emp}]\!]) \cup [\![\mathsf{true}]\!] .$$

Proof. The following logical formula is a pointwise version of the above equation following Section 3.1.

$$\forall\, s \in \textit{Stores}, \forall\, h \in \textit{Heaps} : (\, s, h \models p \Leftrightarrow s, h \models (p \wedge \mathsf{emp}) * \mathsf{true}\,) . \tag{3.9}$$

We show that it is equivalent to the definition given in (3.8). For better readability we omit the universal quantification over stores in the remainder. First, we simplify $s,h \models (p \wedge \mathsf{emp}) * \mathsf{true}$. Using the definitions of Section 2.1, we get for arbitrary $h \in \mathit{Heaps}$

$$\begin{aligned}
& s,h \models (p \wedge \mathsf{emp}) * \mathsf{true} \\
\Leftrightarrow\ & \exists\, h_1,h_2 \in \mathit{Heaps} : \mathit{dom}(h_1) \cap \mathit{dom}(h_2) = \emptyset \wedge h = h_1 \cup h_2 \\
& \wedge\ s,h_1 \models p \ \wedge\ s,h_1 \models \mathsf{emp} \ \wedge\ s,h_2 \models \mathsf{true} \\
\Leftrightarrow\ & \exists\, h_1,h_2 \in \mathit{Heaps} : \mathit{dom}(h_1) \cap \mathit{dom}(h_2) = \emptyset \ \wedge\ h = h_1 \cup h_2 \\
& \wedge\ s,h_1 \models p \ \wedge\ h_1 = \emptyset \\
\Leftrightarrow\ & \exists\, h_2 \in \mathit{Heaps} : h = h_2 \ \wedge\ s,\emptyset \models p \\
\Leftrightarrow\ & s,\emptyset \models p\,.
\end{aligned}$$

Now, Equation (3.8) implies

$$\begin{aligned}
& \forall\, h,h' \in \mathit{Heaps} : (\, s,h \models p \Leftrightarrow s,h' \models p\,) \\
\Rightarrow\ & \forall\, h \in \mathit{Heaps} : (\, s,h \models p \Leftrightarrow s,\emptyset \models p\,) \\
\Leftrightarrow\ & \forall\, h \in \mathit{Heaps} : (\, s,h \models p \Leftrightarrow s,h \models (p \wedge \mathsf{emp}) * \mathsf{true}\,)\,.
\end{aligned}$$

For the converse direction, we conclude for arbitrary s,h,h' using Equation (3.9) and the above result twice that $s,h \models p \Leftrightarrow s,\emptyset \models p \Leftrightarrow s,h' \models p$. □

Hence, any pure assertion p also satisfies the logical formula $p \Leftrightarrow (p \wedge \mathsf{emp}) * \mathsf{true}$ which includes the above mentioned facts that the empty heap and any extension of it also satisfy it. Moreover, the characterisation of Theorem 3.2.5 immediately enables a lifting to the abstracter level of arbitrary Boolean quantales yielding the following result.

Definition 3.2.6
In an arbitrary Boolean quantale S an element a is called *pure* iff it satisfies

$$a = (a \sqcap 1) \cdot \top\,.$$

This equation characterises pure elements as fixed points of the function $f(a) = (a \sqcap 1) \cdot \top$. Since f is built from isotone operations it is also isotone. Now, by the assumptions of an underlying complete lattice, the *Knaster-Tarski* theorem [Tar55] immediately implies that the set of pure elements is also a complete lattice. By the infinite distributivity laws and the assumption of a Boolean quantale, we get for any set of pure elements X that also $\bigsqcup X$ and $\bigsqcap X$ is pure.

As a next step we can use Definition 3.2.6 to give further characterisations of pure elements in arbitrary commutative and Boolean quantales. We list some of them in the following.

Lemma 3.2.7 *In any commutative Boolean quantale, an element a is pure iff one of the following equivalent properties is satisfied.*

(a) $a \cdot \top \leq a$ *and* $\overline{a} \cdot \top \leq \overline{a}$,

(b) $a \cdot \top \leq a$ *and* $a \sqcap b \cdot c \leq (a \sqcap b) \cdot (a \sqcap c)$ *for all* $b, c \in S$,

(c) $(a \sqcap b) \cdot c = a \sqcap b \cdot c$ *for all* $b, c \in S$.

A proof can be found in Appendix A.

The advantages of these characterisations are that part (a) also holds in the setting of non-commutative quantales as it provides a simple characterisation of pureness that immediately reveals that pure elements and their complements are also intuitionistic. We summarise

Corollary 3.2.8 *Every pure element of a Boolean quantale is also intuitionistic.*

In particular, in any Boolean quantale $\overline{0} = \top$ and therefore we can conclude that

Corollary 3.2.9 $\top$ *and* 0 *are the greatest and smallest pure elements, respectively.*

Part (b) of Lemma 3.2.7 states that in addition to being intuitionistic, pure elements also entail an inequational distributivity property. An upper bound of the intersection of an arbitrary product with a pure element a is obtained by multiplying the possible smaller intersections of a with the arguments of the product. Finally, part (c) is stated as a property of pure elements in [Rey02] which is now fully algebraically derived in the abstract setting. By setting $b = \top$ and $c = 1$ it is not difficult to see that Definition 3.2.6 is a special case of it. Note that since the underlying quantale is commutative, it is also possible to use the dual of part (c), namely $b \cdot (a \sqcap c) = a \sqcap b \cdot c$ as a characterisation of pure assertions.

We note that a similar result for a pointfree characterisation of pure assertions has also been derived in [BBTS05, BBTS07]. The characterisation is given within a higher-order logic setting in category theory. It semantically corresponds very much to what is presented in part (c).

Another characterisation for pure elements involves detachments, i.e., in concrete separation logic the septraction operator.

Lemma 3.2.10 *In a commutative Boolean quantale, the distributivity law* $a \sqcap b \cdot c \leq (a \sqcap b) \cdot (a \sqcap c)$ *is equivalent to* $a \lfloor \top \leq a$ *and* $\top \rfloor a \leq a$.

A proof of this lemma is deferred to Appendix A. The element $\top\rfloor a$ can be interpreted in AS as the $\subseteq$-downward closure of a w.r.t. the heap component of states, i.e., by removing parts of the allocated resources in the heap, the respective state will remain in a.

To conclude the paragraph concerning pure elements we list a few properties which can be easily proved completely in our algebraic approach.

Corollary 3.2.11 *Pure elements form a Boolean lattice, i.e., they are closed under* $+$, $\sqcap$ *and* $\overline{\ }$. *Moreover the lattice is complete.*

The following lemma shows that in the complete lattice of pure elements meet and join coincide with composition and sum, respectively.

Lemma 3.2.12 *Consider a commutative Boolean quantale* S, *pure elements* $a, a' \in S$ *and arbitrary elements* $b, c \in S$. *Then*

(a) $a \cdot b = a \sqcap b \cdot \top$,

(b) $a \cdot a' = a \sqcap a'$, *in particular* $a \cdot a = a$ *and* $a \cdot \overline{a} = 0$.

Proof. For a proof of part (a) we calculate, using Lemma 3.2.7(c), $a \sqcap b \cdot \top = a \sqcap \top \cdot b = (a \sqcap \top) \cdot b = a \cdot b$. To show part (b), we use again Lemma 3.2.7(c) and neutrality of 1 w.r.t. multiplication and obtain $a \cdot a' = a \sqcap a' \cdot 1 = a \sqcap a'$. □

Many further properties, in particular, for the interaction of pure assertions with residuals and detachments, can be found in Appendix A.2. It is possible to obtain similar connections for separating implication and septraction interacting with pure assertions like in Lemma 3.2.7(c) where the case for separating conjunction is stated.

We conclude this section with a consideration of pure elements from an algebraic viewpoint. There exists a relationship of pure elements and particular ones that can be found below 1 w.r.t. $\leq$. In the concrete assertion quantale AS, such elements coincide with sets that characterise states involving empty heaps. Generally, these elements are called *tests* [MB85, Koz97] and come with special behaviour which we sum up in the following definition.

Definition 3.2.13 (Tests)
We define a *test* in a quantale S as an element $t \leq 1$ that has a complement $\neg t$ relative to 1, i.e., $t + \neg t = 1$ and $t \cdot \neg t = 0 = t \cdot \neg t$. The set of all tests of S is denoted by $\mathsf{test}(S)$. It is closed under $+$ and $\cdot$, where the former coincides with $\sqcup$ and the latter corresponds to $\sqcap$. Moreover, it forms a Boolean algebra with 0 and 1 as its least and greatest elements.

Note that in any Boolean quantale the element $a \sqcap 1$ for each a is a test where $\neg(a \sqcap 1) = \overline{a} \sqcap 1$ holds for its relative complement. In particular, every element below 1 is a test. Now, consider the characterising equation $a = (a \sqcap 1) \cdot \top$ of pure elements. It exactly renders the relationship of pure elements and test elements in that there exists for each a a corresponding test $a \sqcap 1$. Clearly, the same holds for the complements. In AS such sets of states involve empty heaps and contain exactly all information and assumed conditions on the store variables of a pure assertion. Conversely, for each test $t \leq 1$ there exists also a pure element given by $t \cdot \top$ (cf. Lemma A.2.1 in the Appendix). Note that in any Boolean quantale the complements satisfy $\overline{t \cdot \top} = (\overline{t} \sqcap 1) \cdot \top$ (e.g. [DM01a]).

3.2.3 Preciseness

The previously discussed assertion classes come with the behaviour that assertions are satisfied on some heap and unspecified extensions of it, i.e., they are given in an intuitionistic fashion. However, concrete verification tasks in separation logic frequently require the ability to point out a unique part of the heap. Examples can be found in applications that provide particular frameworks for reasoning about information hiding [ORY09] or that establish special proof rules allowing unspecified sharing within graph structures [HV13].

The concept of *precise assertions* has turned out to be adequate for such reasoning tasks [Rey09]. These assertions ensure the existence of a unique subheap which is relevant to their predicate. A pointwise definition for precise assertions is given by the formula

$$\forall\, s, h, h_1, h_2 : (s, h_1 \models p \wedge s, h_2 \models p \wedge h_1 \subseteq h \wedge h_2 \subseteq h) \Rightarrow h_1 = h_2\,.$$

By this there exists for all states (s, h) at most one subheap h' of h which already contains the allocated resources that p requires, i.e., for which we have $(s, h') \models p$. Concrete examples in separation logic are emp, the single cell assertion $\mathsf{i} \mapsto \mathsf{j}$ for program variables i, j and the recursive list predicate $\mathsf{list}\ \alpha\ \mathsf{i}$ of Example 2.1.1 where α denotes a sequence of values.

Since we are again mainly interested in deriving pointfree formulas for arbitrary Boolean quantales that abstractly reflect the behaviour of this assertion class, we follow a result given in [ORY09]. It is stated there that the above definition for precise assertions is equivalent to the logical formula

$$(p \wedge q) * (p \wedge r) \Leftrightarrow p * (q \wedge r)\,,$$

where p is precise and q, r are arbitrary assertions. The formula states that separating conjunction distributes of over logical conjunction for precise assertions p. Note that

the $\Leftarrow$ - direction is always valid due to logical weakening and thus can be dropped. Applying the previous results on interpreting formulas of separation logic in AS we can immediately state the following theorem.

Theorem 3.2.14 *In* AS *an element* $[\![p]\!]$ *is precise iff it satisfies, for all* $[\![q]\!]$ *and* $[\![r]\!]$,

$$([\![p]\!] \cup [\![q]\!]) \cap ([\![p]\!] \cup [\![r]\!]) \subseteq [\![p]\!] \cup ([\![q]\!] \cap [\![r]\!]) .$$

Hence, for arbitrary Boolean quantales we can algebraically characterise precise assertions as follows.

Definition 3.2.15
In an arbitrary Boolean quantale S an element a is called *precise* iff for all $b, c \in S$

$$a \cdot b \sqcap a \cdot c \leq a \cdot (b \sqcap c) .$$

Obviously, the above inequation can again be strengthened in quantales to an equation by isotony of $\sqcap$ and $\cdot$. In that form it is also called *determinacy* known from relation algebras (e.g., [DM01a]). The above definition algebraically characterises preciseness by distributivity of multiplication over binary infima. A similar characterisation for preciseness within a higher-order logic approach to separation logic can also be found in [BBTS05, BBTS07].

Depending on the used model and application it is also possible to extend Definition 3.2.15 for completeness issues to distributivity over arbitrary non-empty infima like e.g. in [COY07, RG08], i.e.,

$$X \neq \emptyset \Rightarrow \bigsqcap \{a \cdot x : x \in X\} \leq a \cdot \bigsqcap X .$$

In [COY07] the above characterisation involving arbitrary infima is used to abstractly characterise the structure of programs. These are modelled by functions or more concretely state transformers that output strongest postconditions represented as sets of states w.r.t. a given input state satisfying a considered precondition.

For our purposes one can replace the characterisation of Theorem 3.2.14 in AS with a more general one involving arbitrary intersections if required. In the case of separation logic assertions, a concrete instance of such distributivity law can then be found in the formula

$$\forall x.\ (p * q) \Leftrightarrow p * (\forall x.\ q)$$

assuming p is precise and variable x does not occur free in p (cf. [Rey09]).

Since for most purposes the case for binary infima suffices, we stay with the above definition. In particular, this has the advantage that reasoning about preciseness

can be supported by first-order logic theorem proving systems which allow proofs to be derived fully automatically. We continue our algebraic considerations by giving some closure properties for this assertion class which can be proved completely in the algebraic setting.

Lemma 3.2.16 *If a and a' are precise then so is $a \cdot a'$, i.e., precise assertions are closed under multiplication.*

Proof. The proof is by straightforward calculation. For arbitrary elements b, c and precise elements a, a', we have

$$(a \cdot a') \cdot b \sqcap (a \cdot a') \cdot c = a \cdot (a' \cdot b) \sqcap a \cdot (a' \cdot c) \leq a \cdot (a' \cdot b \sqcap a' \cdot c) \leq a \cdot a' \cdot (b \sqcap c).$$

□

Lemma 3.2.17 *If a is precise and $a' \leq a$ then a' is precise, i.e., precise assertions are downward closed.*

Corollary 3.2.18 *For an arbitrary assertion b and precise a, also $a \sqcap b$ is precise.*

Lemma 3.2.19 *a is precise iff $a \cdot \overline{b} \leq \overline{a \cdot b}$ for arbitrary b.*

The latter lemma gives a characterisation of preciseness using Boolean complements. For precise elements it is therefore possible to state some general behaviour that characterises the interaction of multiplication (separating conjunction) and Boolean complementation (logical negation). Proofs of above results are not difficult to obtain and can be found e.g., in [DM01a]. Further useful properties are again listed and proved in Appendix A.2.

3.2.4 Full Allocation

We now turn to the question of whether there exist assertions that satisfy both properties, preciseness and intuitionisticness. At first sight trying to find such assertions does not seem to be sensible for the standard storage model of separation logic given in Section 2.1, since an assertion p that holds for any larger heap cannot unambiguously point out an exact heap portion [Rey09]. However, by the use of the tool MACE4 [McC05] an abstract counter model to this fact was discovered. A reinterpretation of it in separation logic revealed that completely allocated heaps, i.e., heaps that do not allow any further allocation of resources are at the same time precise *and* intuitionistic. Algebraic proofs for this will be provided in the sequel.

Note that this is not contradicting any statement in [Rey09] where allocation of resources is defined there to never abort. The idea to this is that according to Section 2.1

heaps are partial functions with finite domain while the set of addresses is infinite. Hence, for allocation commands there exists always a finite sequence of unallocated addresses that can be used and therefore will be selected non-deterministically.

For the purpose of abstraction the consideration of assertions capturing completely allocated heaps establishes for the presented developments the inclusion of non-standard models, thus making the approach more general. The discovered assertion class with the described property will be called *fully allocated*. They might be helpful in an algebraic treatment for detecting memory leaks in programs. Moreover they provide a simple approach to characterise programs that will eventually abort or show non-deterministic behaviour resulting from any further attempts to allocate heap storage that might not be possible due to an increasing loss of references to allocated resources.

Assuming the set of addresses *Addresses* is finite and the existence of an appropriate definition of the allocation command we can characterise such assertions pointwise by

$$p \text{ is } \textit{fully allocated} \Leftrightarrow_{df} (\forall\, s, h :\ s, h \models p \Rightarrow\ dom(h) = \mathit{Addresses})\,.$$

Theorem 3.2.20 *In* AS *an element* $[\![p]\!]$ *is fully allocated iff it satisfies*

$$[\![p]\!] \cup \overline{[\![\mathsf{emp}]\!]} \subseteq \emptyset\,.$$

Proof.

$$\begin{array}{ll}
 & \forall\, s, h :\ s, h \models p \Rightarrow\ dom(h) = \mathit{Addresses} \\
\Leftrightarrow & \forall\, s, h :\ (s, h \models p \Rightarrow (\forall\, h'.\ h \subseteq h' \Rightarrow h' \subseteq h)) \\
\Leftrightarrow & \forall\, s, h, h' :\ (s, h \models p \Rightarrow (h \subseteq h' \Rightarrow h' \subseteq h)) \\
\Leftrightarrow & \forall\, s, h, h' :\ (s, h \models p \Rightarrow \neg(h \subseteq h' \wedge h' - h \neq \emptyset)) \\
\Leftrightarrow & \forall\, s, h' :\ \neg(\exists\, h :\ s, h \models p \wedge h \subseteq h' \wedge h' - h \neq \emptyset) \\
\Leftrightarrow & \forall\, s, h' :\ \neg(\exists\, h, h'' :\ s, h \models p \wedge h'' \neq \emptyset \wedge\ dom(h) \cap dom(h'') = \emptyset \\
 & \qquad\qquad \wedge h' = h \cup h'') \\
\Leftrightarrow & \forall\, s, h' :\ s, h' \models p * \neg\mathsf{emp} \Rightarrow \mathsf{false}\,.
\end{array}$$

□

Consequently in the quantale-based algebraic setting we can characterise this class as follows.

Definition 3.2.21
In an arbitrary Boolean quantale S an element a is called *fully allocated* iff

$$a \cdot \overline{1} \leq 0\,.$$

Lemma 3.2.22 *Every fully allocated element is also intuitionistic.*

Proof. Let a be a fully allocated element, then $a \cdot \top = a \cdot (1 + \overline{1}) = a \cdot 1 + a \cdot \overline{1} \leq a \cdot 1 = a$. These (in)equations hold by Boolean algebra, distributivity, p being fully allocated and neutrality of 1 w.r.t. multiplication. □

Lemma 3.2.23 *If a is fully allocated then a is also precise.*

For the proof we need an auxiliary property.

Theorem 3.2.24 *If a is fully allocated then $a \cdot b = a \cdot (b \sqcap 1)$ holds.*

Proof. We calculate

$$a \cdot b = a \cdot ((b \sqcap 1) + (b \sqcap \overline{1})) = a \cdot (b \sqcap 1) + a \cdot (b \sqcap \overline{1}) = a \cdot (b \sqcap 1)$$

which follows from Boolean algebra, distributivity and since $a \cdot (b \sqcap \overline{1}) \leq a \cdot \overline{1} \leq 0$ by isotony of $\cdot$ and the assumption. □

Intuitively, adding extra storage to the heap of fully allocated element a is only possible if it is empty. In particular, by this only the store component of a is affected by b.

We note again that in any Boolean quantale S elements of the form $a \sqcap 1$ are tests (cf. Definition 3.2.13). Moreover, according to [Möl07] we have

$$t \cdot (a \sqcap b) = t \cdot a \sqcap b = t \cdot a \sqcap t \cdot b\,. \tag{testdist}$$

for arbitrary tests t and elements $a, b \in S$. And as a direct consequence, one gets validity of the equation $t_1 \cdot a \sqcap t_2 \cdot a = t_1 \cdot t_2 \cdot a$ for any tests t_1, t_2 and arbitrary element $a \in S$.

Now we are ready to show Lemma 3.2.23.

Proof of 3.2.23. For a fully allocated element a and arbitrary b and c, we get

$$a \cdot b \sqcap a \cdot c \;=\; a \cdot (b \sqcap 1) \sqcap a \cdot (c \sqcap 1) \;=\; a \cdot (b \sqcap 1) \cdot (c \sqcap 1) \leq a \cdot (b \sqcap c)\,.$$

Again this holds by applying Theorem 3.2.24 twice, the consequence of (testdist), isotony of $\cdot$, $\sqcap$ and the fact that multiplication coincides with $\sqcap$ on test elements. □

3.2.5 Supported Assertions

The last class of assertions we turn to is the set of so-called *supported* assertions. They ensure that the set of subheaps that satisfy such an assertion has a least element. This

condition is more liberal compared to preciseness. Generally, these assertions also establish the often required full distributivity property of $*$ over $\wedge$ with a restriction of all other occurring assertions to intuitionistic ones. A simple point-free proof of this will be presented later.

We start with a pointwise definition of supported assertions which can be found e.g., in [Rey09]. An assertion p is called *supported* iff

$$\begin{aligned}\forall\, s, h_1, h_2 :\ & h_1, h_2 \text{ are compatible} \wedge s, h_1 \models p \wedge s, h_2 \models p \\ & \Rightarrow\ \exists\, h' :\ h' \subseteq h_1 \wedge h' \subseteq h_2 \wedge s, h' \models p\,.\end{aligned}$$

By assuming that h_1 and h_2 are compatible, it is meant that they agree on their intersection, i.e., $h_1 \cup h_2$ is a partial function again. For an algebraic characterisation of supported elements in arbitrary Boolean quantales we first show the following result.

Theorem 3.2.25 *In* AS *an element* $[\![p]\!]$ *is supported iff it satisfies, for all set of states* $[\![q]\!]$ *and* $[\![r]\!]$*,*

$$([\![p]\!] \cup [\![q]\!]) \cap ([\![p]\!] \cup [\![r]\!]) \subseteq [\![p]\!] \cup ([\![q]\!] \cup [\![\mathsf{true}]\!] \cap [\![r]\!] \cup [\![\mathsf{true}]\!])\,.$$

This inequation is similar to the characterisation of preciseness except that the right-hand side is weakened to the intersection of ideals or vectors. The binary case can also be generalised to arbitrary non-empty (infinite) intersections, i.e., a complete characterisation depending on the application and model used.

The key idea to prove Theorem 3.2.25 is to use special assertions p which describe a set $[\![p]\!]$ that contains exactly a single state (s, h). For this we use the denotation $[\![(s,h)]\!] = \{(s,h)\}$. The proof requires an auxiliary lemma with some simple properties of the predicate (s, h).

Lemma 3.2.26 *Assume arbitrary heaps* h, h' *and store* s *then*

(a) $s, h' \models (s,h) \Leftrightarrow h = h'$. *In particular,* $s, h \models (s,h)$.

(b) *If* $h \subseteq h'$ *then* $s, h \models p \Leftrightarrow s, h' \models p * (s, h' - h)$ *for any assertion* p.

(c) $s, h' \models (s,h) * \mathsf{true} \Leftrightarrow h \subseteq h'$.

The lengthy, but straightforward proof can be found in Appendix A. We continue with a proof of Theorem 3.2.25.

Proof of Theorem 3.2.25. For the $\Rightarrow$ - direction we assume p is supported. Let $s, h \models p * q \wedge p * r$. Then

$$
\begin{array}{ll}
& s,h \models p * q \wedge p * r \\
\Leftrightarrow & \quad \{\![\text{ definition of } *]\!\} \\
& \exists\, h_1, h_2 : h_1 \subseteq h \wedge h_2 \subseteq h \wedge s, h_1 \models p \wedge s, h - h_1 \models q \\
& \qquad\qquad \wedge\, s, h_2 \models p \wedge s, h - h_2 \models r \\
\Rightarrow & \quad \{\![\, p \text{ supported, } h_1 \cup h_2 \subseteq h \text{ is a function }]\!\} \\
& \exists\, h_1, h_2, h' : s, h' \models p \wedge h' \subseteq h_1 \subseteq h \wedge h' \subseteq h_2 \subseteq h \\
& \qquad\qquad \wedge\, s, h - h_1 \models q \wedge s, h - h_2 \models r \\
\Rightarrow & \quad \{\![\, h - h_1 \subseteq h - h', \, s, h_1 - h' \models \mathsf{true}, \text{ analogously } h_2 \,]\!\} \\
& \exists\, h' : s, h' \models p \wedge s, h - h' \models q * \mathsf{true} \wedge s, h - h' \models r * \mathsf{true} \\
\Leftrightarrow & \quad \{\![\text{ definition of } *]\!\} \\
& s, h \models p * (q * \mathsf{true} \wedge r * \mathsf{true}) \,.
\end{array}
$$

For the other direction we assume, for arbitrary assertions q, r and states (s, h) that

$$
s, h \models p * q \wedge p * r \;\Rightarrow\; s, h \models p * (q * \mathsf{true} \wedge r * \mathsf{true}) \tag{3.10}
$$

as well as $s, h_1 \models p$ and $s, h_2 \models p$ for arbitrary heaps h_1, h_2 with $h_1 \cup h_2$ is a function. From this we calculate

$$
\begin{array}{ll}
& s, h_1 \models p \wedge s, h_2 \models p \\
\Leftrightarrow & \quad \{\![\, h_1 \subseteq h_1 \cup h_2 \text{ and } h_2 \subseteq h_1 \cup h_2 \text{ and Lemma 3.2.26(b) }]\!\} \\
& s, h_1 \cup h_2 \models p * (s, (h_1 \cup h_2) - h_1) \wedge s, h_1 \cup h_2 \models p * (s, (h_1 \cup h_2) - h_2) \\
\Rightarrow & \quad \{\![\text{ Assumption (3.10) }]\!\} \\
& s, h_1 \cup h_2 \models p * \big((s, (h_1 \cup h_2) - h_1) * \mathsf{true} \wedge (s, (h_1 \cup h_2) - h_2) * \mathsf{true}\big) \\
\Leftrightarrow & \quad \{\![\text{ definition of } *]\!\} \\
& \exists\, h' : h' \subseteq h_1 \cup h_2 \wedge s, h' \models p \wedge \\
& \qquad s, (h_1 \cup h_2) - h' \models (s, (h_1 \cup h_2) - h_1) * \mathsf{true} \wedge (s, (h_1 \cup h_2) - h_2) * \mathsf{true} \\
\Leftrightarrow & \quad \{\![\text{ definition of } \wedge \,]\!\} \\
& \exists\, h' : h' \subseteq h_1 \cup h_2 \;\wedge s, h' \models p \\
& \qquad \wedge\, s, (h_1 \cup h_2) - h' \models (s, (h_1 \cup h_2) - h_1) * \mathsf{true} \\
& \qquad \wedge\, s, (h_1 \cup h_2) - h' \models (s, (h_1 \cup h_2) - h_2) * \mathsf{true} \\
\Leftrightarrow & \quad \{\![\text{ Lemma 3.2.26(c) (twice) }]\!\} \\
& \exists\, h' : h' \subseteq h_1 \cup h_2 \wedge s, h' \models p \wedge (h_1 \cup h_2) - h_1 \subseteq (h_1 \cup h_2) - h' \\
& \qquad \wedge\, (h_1 \cup h_2) - h_2 \subseteq (h_1 \cup h_2) - h' \\
\Leftrightarrow & \quad \{\![\text{ by } h' \subseteq h_1 \cup h_2 \wedge (h_1 \cup h_2) - h_i \subseteq (h_1 \cup h_2) - h' \Rightarrow h' \subseteq h_i \,]\!\} \\
& \exists\, h' : h' \subseteq h_1 \cup h_2 \wedge s, h' \models p \wedge h' \subseteq h_1 \wedge h' \subseteq h_2 \\
\Leftrightarrow & \quad \{\![\text{ logic step }]\!\} \\
& \exists\, h' : s, h' \models p \wedge h' \subseteq h_1 \wedge h' \subseteq h_2 \,.
\end{array}
$$

□

As for the assertion classes before, this characterisation can be lifted to the abstract level of quantales.

Definition 3.2.27
In a Boolean quantale S an element a is *supported* iff it satisfies for arbitrary b,c

$$a \cdot b \sqcap a \cdot c \leq a \cdot (b \cdot \top \sqcap c \cdot \top) .$$

Following this characterisation of supported assertions we can now abstractly derive various useful properties as direct consequences in a completely algebraic fashion.

Lemma 3.2.28 *If a is supported and b, c are intuitionistic then*

$$a \cdot b \sqcap a \cdot c \leq a \cdot (b \sqcap c) .$$

Proof. Immediately from the definitions. □

Again this result describes the common usage of supported assertions. They are less strict than precise ones and at the same time enable the frequently required distributivity law.

Lemma 3.2.29 *If a is pure then it is also supported. In particular, 0 and $\top$ are supported.*

Proof. By Lemma 3.2.12(a) twice, associativity, commutativity, idempotence of $\sqcap$, isotony and Lemma 3.2.12(a) again,

$$\begin{aligned} a \cdot b \sqcap a \cdot c &= (a \sqcap b \cdot \top) \sqcap (a \sqcap c \cdot \top) \\ &= a \sqcap (b \cdot \top \sqcap c \cdot \top) \\ &\leq a \sqcap (b \cdot \top \sqcap c \cdot \top) \cdot \top \\ &= a \cdot (b \cdot \top \sqcap c \cdot \top) . \end{aligned}$$

□

For pure assertions a least subheap is identified with the empty heap and hence it is clear that they are also supported.

Lemma 3.2.30 *a is precise implies a is supported.*

Proof. By the definition of precise elements and isotony we infer

$$a \cdot b \sqcap a \cdot c \leq a \cdot (b \sqcap c) \leq a \cdot (b \cdot \top \sqcap c \cdot \top) .$$

□

Lemma 3.2.31 *Supported elements are closed under* $\cdot$.

Proof. Assume supported elements a and a' then

$$\begin{aligned} a \cdot a' \cdot b \sqcap a \cdot a' \cdot c &\leq a \cdot (a' \cdot b \cdot \top \sqcap a' \cdot c \cdot \top) \\ &\leq a \cdot a' \cdot (b \cdot \top \cdot \top \sqcap c \cdot \top \cdot \top) \\ &\leq a \cdot a' \cdot (b \cdot \top \sqcap c \cdot \top)\,. \end{aligned}$$

□

Corollary 3.2.32 *If a is supported and b is precise or pure then $a \cdot b$ is supported.*

Lemma 3.2.33 *$a \cdot \top$ is supported implies that also a is supported.*

Proof. Assuming a is supported, we infer from isotony, commutativity and $\top \cdot \top = \top$

$$a \cdot b \sqcap a \cdot c \leq a \cdot \top \cdot (b \cdot \top \sqcap c \cdot \top) \leq a \cdot (\top \cdot b \cdot \top \sqcap \top \cdot c \cdot \top) = a \cdot (b \cdot \top \sqcap c \cdot \top)\,.$$

□

In summary, we can conclude that

Corollary 3.2.34 *a is supported iff $a \cdot \top$ is supported.*

Next we continue with a so-called *precising operation* (e.g. in [Rey08]) introduced by Yang. Is used to clarify the concrete relationships of precise assertions with the ones being at the same time supported and intuitionistic. The operation is defined in separation logic by the mapping $\mathsf{Pr}(p) =_{df} p \wedge \neg(p * \neg\,\mathsf{emp}\,)$. Intuitively, it removes all non-empty resources of states that are not required for satisfying an assertion p. Since $\mathsf{Pr}(\_)$ is already given in a pointfree form we can immediately abstract it to the setting of commutative Boolean quantales S by

$$\mathsf{Pr}(a) = a \sqcap \overline{a \cdot \overline{1}}$$

for an element $a \in S$. In the sequel we will require additional assumptions for managing algebraic proofs of properties for this mapping. The assumptions characterise specific behaviour of the carrier set *States* but are still general and natural for reasoning about resources.

We begin with a special property for the emptiness assertion emp. It reads in separation logic $\neg\mathsf{emp} * \neg\mathsf{emp} \Rightarrow \neg\mathsf{emp}$ and intuitively states that each heap that can be split into at least two non-empty subheaps still remains non-empty. For arbitrary Boolean quantales this abstracts to

$$\overline{1} \cdot \overline{1} \leq \overline{1} \qquad \text{(nonemp)}$$

and does not generally hold. As a consequence of this we summarise

Lemma 3.2.35 $\overline{1} \cdot \top = \overline{1}$ *and* $1 \leq \overline{\overline{1} \cdot \overline{1}}$.

Proof. First, by Boolean algebra, distributivity, neutrality and (nonemp)

$$\overline{1} \cdot \top = \overline{1} \cdot (1 + \overline{1}) = \overline{1} \cdot 1 + \overline{1} \cdot \overline{1} = \overline{1}\,.$$

The second inequation follows from shunting. □

Hence, assuming (nonemp) also $\overline{1}$ turns into an intuitionistic element. In the literature, the element $\overline{\overline{1} \cdot \overline{1}}$ has also been used in the context of temporal logics [VK98, Höf09] and is called there step. Although it is interpreted by progress in time, results of these works are abstractly provided within an algebraic approach and hence can be transferred and reinterpreted into a separation logical setting.

We continue with pointfree proofs of properties related to $\mathsf{Pr}(\_)$ which can also be found in [Rey08].

Lemma 3.2.36 $\mathsf{Pr}(1 + a) = 1$. *In particular,* $\mathsf{Pr}(\top) = 1$.

Proof. By definition of $\mathsf{Pr}(\_)$, Boolean algebra and distributivity

$$\begin{aligned}\mathsf{Pr}(1 + a) &= (1 + a) \sqcap \overline{(1 + a) \cdot \overline{1}} = (1 + a) \sqcap \overline{1 \cdot \overline{1} + a \cdot \overline{1}} \\ &= (1 + a) \sqcap (1 \sqcap \overline{a \cdot \overline{1}}) = (1 \sqcap \overline{a \cdot \overline{1}}) + (a \sqcap 1 \sqcap \overline{a \cdot \overline{1}}) \\ &= 1 \sqcap \overline{a \cdot \overline{1}}\,.\end{aligned}$$

Clearly, $\mathsf{Pr}(1 + a) \leq 1$. For the other direction we calculate using Lemma 3.2.35 that $1 = 1 \sqcap \overline{\overline{1}} = 1 \sqcap \overline{\top \cdot \overline{1}} \leq 1 \sqcap \overline{a \cdot \overline{1}}$. □

Lemma 3.2.37 *If* a *is precise then* $\mathsf{Pr}(a) = a$.

Proof. Setting $b = 1$ in Lemma 3.2.19 one obtains $a \cdot \overline{1} \leq \overline{a}$. This is equivalent to $a \leq \overline{a \cdot \overline{1}}$ and hence $\mathsf{Pr}(a) = a \sqcap \overline{a \cdot \overline{1}} = a$. □

Lemma 3.2.38 *If* a *is precise then* $\mathsf{Pr}(a \cdot \top) = a$.

Proof. Lemma 3.2.35 simplifies $a \cdot \top \sqcap \overline{a \cdot \top \cdot \overline{1}} = a \cdot \top \sqcap \overline{a \cdot \overline{1}}$. By Boolean algebra we infer $a \cdot \top \sqcap \overline{a \cdot \overline{1}} = (a + a \cdot \overline{1}) \sqcap \overline{a \cdot \overline{1}} = a \sqcap \overline{a \cdot \overline{1}} \leq a$. Lemma 3.2.19 implies the converse $a = a \sqcap a \cdot \top \leq a \cdot \top \sqcap \overline{a \cdot \overline{1}}$. □

This means that the precising operation does not modify precise elements. Moreover, note that by Lemma 3.2.30 and Corollary 3.2.34 elements of the form $a \cdot \top$ are intuitionistic and supported for a precise a. Hence $\mathsf{Pr}(\_)$ turns supported and intuitionistic elements built from precise ones back into that original form. In [Rey08]

it is shown in separation logic that the logical mapping $(\_) * \mathsf{true}$ conversely has the same behaviour on assertions $\mathsf{Pr}(p)$ if p is intuitionistic and supported.

Unfortunately an algebraic proof of this fact requires again an additional assumption for a property of supported assertions that can not be generally derived in arbitrary Boolean quantales from Definition 3.2.27. A reason for this is that the abstraction to quantales also includes concrete algebras besides AS that come with different behaviour excluding such specific properties. It is therefore necessary to restrict the setting to a subset of algebras to manage the required properties.

The concrete property is formulated in [Rey08] by a result that states that the definition for supported assertions p is equivalent to assuming that for arbitrary s, h: whenever $\{h' : h' \subseteq h, (s, h') \models p\} \neq \emptyset$ then it has a least element. The inclusion of this behaviour within the algebraic approach requires that supported elements a additionally satisfy

$$a \leq (a \sqcap \overline{a \cdot \bar{1}}) \cdot \top \,. \tag{suppleast}$$

Intuitively this means in separation logic that heap cells characterised by $\mathsf{Pr}(p)$ are contained in any heap of states characterised by supported assertions p. In the concrete model, it also represents the least heap that satisfies p. To see this we give Equation (suppleast) in a pointwise form for arbitrary s, h

$$\begin{aligned} s, h \models p \;\Rightarrow\; & \exists\, h_1 : h_1 \subseteq h \wedge s, h_1 \models p \wedge \\ & (\forall\, h' : h' \subseteq h_1 \wedge s, h' \models p \;\Rightarrow\; h_1 - h' = \emptyset)\,. \end{aligned}$$

Instead of assuming (suppleast) in addition to Definition 3.2.27, another possibility would be to use an equivalent characterisation for supported elements that involves $\mathsf{Pr}(\_)$ and implies (suppleast).

Lemma 3.2.39 *a is supported and satisfies* (suppleast) *iff* $a \cdot b \sqcap a \cdot c \leq \mathsf{Pr}(a) \cdot (b \cdot \top \sqcap c \cdot \top)$.

Proof. The $\Rightarrow$-direction follows from isotony, assumptions and $\top \cdot (b \cdot \top \sqcap c \cdot \top) \leq \top \cdot b \cdot \top \sqcap \top \cdot c \cdot \top = b \cdot \top \sqcap c \cdot \top$. For the reverse implication we infer by $\mathsf{Pr}(a) \leq a$ that a is supported and setting $b = c = 1$ yields (suppleast). □

This means in particular that under the assumption of (suppleast) the original algebraic definition of supported elements is equivalent to the new one involving the precising operation. Finally, we conclude

Lemma 3.2.40 $\mathsf{Pr}(a) \cdot \top = a$ *when a satisfies* (suppleast) *and is intuitionistic.*

Proof. By isotony and a is intuitionistic, we have $\mathsf{Pr}(a) \cdot \top \leq a \cdot \top \leq a$. Moreover, the other inequation follows from (suppleast). □

As future work on this topic, it would be interesting to investigate the algebraic calculus whether there exists a more suitable axiomatisation that represents an adequate abstraction of supported assertions, especially for the case of quantales similar to AS. Lemma 3.2.39 can be used as a starting point but unfortunately is not suitable for obtaining simple and concise derivations of properties. There still exist properties for which it is not known if they can be shown from that characterisation. This could be due to the lack of laws for characterising an interplay of multiplication and complementation or due to undecidability results on propositional separation logic [BK10]. The conclusions of latter approach are formalised within the context of proof systems for BI logics. Corresponding algebras are closely related to the algebraic approach based on quantales (cf. Section 3.1.1). There might also be some relationships to [BV14]. An extension of Boolean BI algebras is introduced there that allows the characterisation of basic and frequently required properties within the algebra which was not possible using the former approach.

Example properties which we were not able to infer are closure of the characterisation in Lemma 3.2.39 under multiplication like in Lemma 3.2.31, where it is shown under the assumption of Definition 3.2.27. Furthermore, it is not known if Lemma 3.2.39 implies in arbitrary commutative Boolean quantales that $\mathsf{Pr}(a)$ is precise assuming that a is supported as stated in [Rey08]. Only a stronger result stating that $\mathsf{Pr}(a)$ is supported can be immediately inferred.

3.3 Relationship to Separation Algebras

We showed that a large part of the assertions and their behaviour in separation logic can be lifted to a point-free and algebraic setting that allowed simple proofs of non-trivial properties in a calculational style due to the abstraction from irrelevant details. The obtained (in)equations of the previous sections generally reflect the abstract behaviour of the specific assertion classes without depending on structural properties of the concrete considered model. Hence the characterisations can also be used for other separation logics.

Another approach that abstracts from the concrete definition of states as pairs of stores and heaps has been taken by Calgagno and others in [COY07]. In their work, states are seen as arbitrary resources of a program which themselves come with an algebraic structure. They form elements of a so-called *separation algebra* or resource algebra. Assertions are given like in AS using a powerset structure, i.e., as sets of states.

In the following, we give a definition and concrete examples of such algebras. Moreover, we relate the standard store and heap model of Section 2.1 with such structures

and explain how the results on pointfree algebraic characterisations for assertions can be interpreted for resource algebras.

Definition 3.3.1 (Separation algebra [COY07])
A *separation algebra* is a cancellative and partial commutative monoid that we denote by $(\Sigma, \bullet, u)$. Elements of the algebra are called *states* or *resources* and denoted by $\sigma, \tau, \ldots \in \Sigma$. Due to partiality two terms are defined to be equal iff both are defined and equal or both terms are undefined. This induces a *combinability* relation $\#$ defined by

$$\sigma_0 \# \sigma_1 \;\Leftrightarrow_{df}\; \sigma_0 \bullet \sigma_1 \text{ is defined}$$

and a *substate* relation given for $\sigma_0, \sigma_1 \in \Sigma$ by

$$\sigma_0 \preceq \sigma_1 \;\Leftrightarrow_{df}\; \exists\, \sigma_2.\ \sigma_0 \bullet \sigma_2 = \sigma_1\,.$$

By writing $\sigma \bullet \tau$ for states σ, τ we will implicitly assume $\sigma \# \tau$ in the sequel if it is not explicitly stated. The *empty state* u is the unit of the partial binary operator $\bullet$ which satisfies cancellativity in the sense that $\sigma_1 \bullet \tau = \sigma_2 \bullet \tau \;\Rightarrow\; \sigma_1 = \sigma_2$ for arbitrary states σ_1, σ_2, τ.

Next we provide some concrete examples for separation algebras to get an idea for applications of those structures.

Example 3.3.2

a) A standard example is given by the set of heaps represented as partial functions. The corresponding separation algebra is denoted by $(\mathit{Heaps}, \cup, \emptyset)$ where combinability for heaps h_0, h_1 is defined by

$$h_0 \# h_1 \;\Leftrightarrow\; \mathit{dom}(h_0) \cap \mathit{dom}(h_1) = \emptyset\,.$$

Clearly, $\cup$ is used in this context as the disjoint union of partial functions as in the standard carrier set *States*. For the purpose of abstraction one can deviate to arbitrary sets of addresses and values instead of using the instances *Addresses* and *Values* of Section 2.1.

b) Another simple model is the algebra $(\mathit{States}, *_S, \{(\emptyset, \emptyset)\})$ where

$$(s_0, h_0) *_S (s_1, h_1) =_{df} (s_0 \cup s_1, h_0 \cup h_1)\,.$$

By this definition, the involved stores are also treated like heaps, i.e.,

$$(s_0, h_0) \# (s_1, h_1) \;\Leftrightarrow\; \mathit{dom}(s_0) \cap \mathit{dom}(s_1) = \emptyset \;\wedge\; \mathit{dom}(h_0) \cap \mathit{dom}(h_1) = \emptyset\,.$$

The effect of this combinability relation is that using a lifted version of $*_S$ to assertions, store variable definitions are only kept locally. Hence assignments to variables are not globally visible. In sum, states in this model can only be unified iff the domains of partial functions modelling stores and heaps are disjoint.

□

Example 3.3.3 (Permission Algebras)
More interestingly, heaps as partial functions can also be extended to carry for each of their cells an additional value that signalises if reading and/or writing to that cell is permitted [Boy03, BCOP05]. Applications for such algebras can be found e.g., in approaches involving concurrency [Vaf11]. Permissions allow heaps to overlap on some of their addresses, i.e., sharing parts of their resources. The values modelling permission form an algebraic structure called *permission algebra*, given by a partial commutative semigroup $(P, \star)$. For this separation algebra heaps are defined as partial functions $A \rightharpoonup (V \times P)$ that map a set of addresses A to pairs consisting of a value of V and an element of a permission algebra P. Combinability is given by

$$h_0 \# h_1 \;\Leftrightarrow\; h_0(a) \star h_1(a) \text{ is defined for all } a \in dom(h_0) \cap dom(h_1)$$

where $\star$ is lifted to $V \times P$ by

$$(v_0, p_0) \star (v_1, p_1) =_{df} \begin{cases} (v_0, p_0 \star p_1) & v_0 = v_1 \wedge p_0 \star p_1 \text{ is defined} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Hence all overlapping addresses of the considered heaps need to agree on the values in V and have combinable permissions in P. In sum, the combinator $\cup_\star$ for heaps h_0, h_1 is defined by

$$(h_0 \cup_\star h_1)(a) =_{df} \begin{cases} h_0(a) & a \in dom(h_0) - dom(h_1) \\ h_1(a) & a \in dom(h_1) - dom(h_0) \\ h_0(a) \star h_1(a) & \text{otherwise.} \end{cases}$$

The basic idea is that shared heap cells that might belong to different threads of a program should not be changed arbitrarily to exclude inconsistencies and therefore non-deterministic program behaviour. Depending on the permission values involved either read and write access or only read access is granted to certain threads. For a better intuition we provide some prominent examples of permission algebras that can be found the literature.

Fractional permissions [Boy03] are rational values v in the interval $(0, 1]$. They are interpreted as follows: If $v = 1$, full permission to read and write a particular cell is given while any value < 1, only grants read access. Concretely, one has $v_0 \star v_1 = v_0 + v_1$

and clearly this is only defined iff $v_0 + v_1 \leq 1$. The advantage of this approach is that full permission to each cell can be split infinitely often for transferring read accesses to arbitrarily many threads.

Counting permissions [BCOP05] are used to keep track of the set of threads that maintain read permission to a resource. In this setting each allocated resource keeps the value $v = 0$ at the beginning which grants total permission. Now, read access is transferred to a thread which receives the value -1 while v gets increased by 1 and thus v denotes the number of read permissions that has been split off. Summarised, counting permissions are elements of $\mathbb{Z}$ and for integers i, j the combination $i \star j = i + j$ is defined iff either $i < 0 \wedge j < 0$ or when one of the values is ≥ 0 then also $i + j \geq 0$. Intuitively for each resource there exists only one positive permission and it is not possible that more read permissions have been transferred than tracked in the positive one. □

More examples of separation and permission algebras can be found in [COY07]. Generally, this formalism captures a variety of interesting models. An exploration of the concrete relationships to the quantale-based treatment of separation logic assertions would immediately extend the application range of that approach.

First, it can be seen that the standard model used in separation logic that used store-heaps pairs as states is not listed in the examples above. Instead, one either misses the store component or it is treated in the same way as the heap, i.e., it is split into disjoint parts (cf. Example 3.3.2). In fact, if we would define the $\cup$ of Section 3.1 on states by

$$(s, h) \cup (s', h') =_{df} (s, h \cup h) \Leftrightarrow s = s' \wedge dom(h) \cap dom(h') = \emptyset$$

that model would not form a separation algebra according to Definition 3.3.1 since $\cup$ would not have a unit element in *States*. It is possible to find a unit for a subset of states by a restriction to a fixed store s in $S(s) =_{df} \{(s, h) : h \in Heaps\}$. The unit w.r.t. $\cup$ would then be $(s, \emptyset)$. Hence, by the only consideration of states with equal stores in $\cup$ an asymmetry in the treatment of resources within a separation algebra is introduced. Such a resource model would correspond to a so-called *multi-unit* separation algebra as defined in [DHA09]. Such algebras come with the same axioms as in Definition 3.3.1 with the difference that there exists a set of units U satisfying $\forall \sigma \in \Sigma : \exists u_i \in U : u_i \bullet \sigma = \sigma$. Hence the standard model on the carrier set *States* can be seen as the union of *single-unit* separation algebras as provided in Definition 3.3.1. Each of these individual single-unit separation algebras can be distinguished by its store s.

Following [DHA09] there exists a possibility to get the standard model into the form of single-unit separation algebra. This is abstractly provided by the definition of a

lifting operator for multi-unit separation algebras. Its basic idea is to remove all unit elements and replace them by a new distinct element that will represent the unit of the resulting separation algebra. Concretely, in the particular case of store and heap pairs, we do this by restructuring the carrier set *States*. We define a particular set to include all units while all other states become singleton sets and further redefine $\uplus$ on these elements. Concretely, we define the lifted setting by

$$\begin{array}{rcl} \mathit{LStates} & =_{df} & \{\{(s,h)\} : (s,h) \in \mathit{States},\, h \neq \emptyset\}\,, \\ \mathsf{emp}_L & =_{df} & \{(s,\emptyset) : s \in \mathit{Stores}\} \end{array}$$

and additionally set for $p, q \in \mathit{LStates} \cup \{\mathsf{emp}_L\}$

$$\begin{array}{rcl} p \uplus q & =_{df} & \{(s, h \cup h') : (s,h) \in p \wedge (s,h') \in q\}\,, \text{ where} \\ p \,\#\, q & \Leftrightarrow & \exists\, (s,h) \in p\,,\, (s',h') \in q :\; \mathit{dom}(h) \cap \mathit{dom}(h') = \emptyset \,\wedge\, s = s'\,. \end{array}$$

It is not difficult to verify that $p \uplus \mathsf{emp}_L = p = \mathsf{emp}_L \uplus p$ for arbitrary $p \in \mathit{LStates} \cup \{\mathsf{emp}_L\}$. In the case of $p \neq \mathsf{emp}_\mathsf{L}$ the resulting set is again a singleton set and otherwise equals emp_L. Moreover, $\uplus$ remains commutative, associative and clearly satisfies cancellativity. Hence, the structure $(\mathit{States}_\mathsf{L} \cup \{\mathsf{emp}_\mathsf{L}\}, \uplus, \mathsf{emp}_\mathsf{L})$ forms a separation algebra and thus can be used within treatments that use this form of abstraction.

Now, for modelling assertions using separation algebras one can use a straightforward lifting to sets of elements, like in Section 3.1. This is done in the powerset model given in [COY07]. Assuming a separation algebra $(\Sigma, \bullet, u)$, the carrier set is given by $\mathcal{P}(\Sigma)$ and one defines for $P, Q \in \mathcal{P}(\Sigma)$

$$\begin{array}{rcl} P * Q & =_{df} & \{\sigma \bullet \tau : \sigma \,\#\, \tau,\, \sigma \in P,\, \tau \in Q\}\,, \\ \mathsf{emp} & =_{df} & \{u\}\,. \end{array}$$

By similar argumentations as given in the proof of Theorem 3.1.2, $(\mathcal{P}(\Sigma), \subseteq, *, \mathsf{emp})$ also forms a Boolean commutative quantale. Moreover, due to the abstract behaviour of the pointfree characterisations, the results on the assertion classes can simply be applied for this particular model and thus for separation algebras.

For obtaining corresponding pointwise formulas interpreted on abstract resource algebras, one can identify occurrences of states (s,h) and single heaps h with adequate elements of an underlying separation algebra $(\Sigma, \bullet, u)$. A formulation of preciseness can be found e.g., in [COY07].

Chapter 4

Relational Separation

An algebra for the program part of separation logic is developed based on the calculus of relations. For this we extend the relational structure with a connective to adequately model separation that in particular allows a further reuse of abstract derived results on the assertion part. Moreover relational interpretations of program commands and formulations of abstract behaviour are provided. This enables pointfree soundness proofs of the frame rule in a calculational and concise way. Moreover, we give further general formulations and relationships to approaches that involve concurrency like concurrent separation logic and concurrent Kleene algebras. Finally, we derive by the use of the relational structure compositional and pointfree abstractions for the framework of dynamic frames.

4.1 Interpreting Commands Relationally

As a starting point for setting up a calculus based on relations for separation logic we begin with deriving a relational interpretation of all commands from the small-step operational semantics given in Section 2.2. For managing this, one obtains from the semantics the general effects of each execution of a single command starting from an arbitrary configuration by relating any adequate input state with some altered output one. We generally follow the approach of [DHM11] to model program commands as relations between states and provide interpretations for the case of a partial and total correctness setting as in [DGM+14]. Clearly, since the latter additionally requires that termination needs to be ensured, the concrete treatment of such commands

differs from the partial correctness one. Moreover, note that the basic assumptions for establishing validity of the frame rule distinguish non-termination from program abortion which results from prohibited memory access (cf. Section 2.3). Hence for an abstract treatment it is also required to include the additional state abort in the partial correctness case while it is not needed for a treatment in a total correctness setting since program abortion and non-termination are identified and interpreted as the empty relation $\emptyset$ there.

Definition 4.1.1 (Relational commands)
A *command* in a total correctness setting is a relation

$$C \in \mathit{TCmds} \;=_{df}\; \mathcal{P}(\mathit{States} \times \mathit{States})\,.$$

In the case of partial correctness it is given by

$$C \in \mathit{PCmds} \;=_{df}\; \mathcal{P}((\mathit{States} \times (\mathit{States} \cup \{\,\mathsf{abort}\,\})) \cup \{(\,\mathsf{abort}\,,\,\mathsf{abort}\,)\})$$

where $\mathsf{abort} \notin \mathit{States}$ represents a distinct state.

Note that commands in PCmds that start from abort will remain in that state. By this there can not exist a transition to some state (s,h) starting from an erroneous one, i.e., the program will get stuck whenever a memory error occurs in the program execution. Both settings yield a denotational model for the commands of separation logic by inductively assigning a formal semantics, i.e., a relation $[\![\, C \,]\!] \in \mathit{PCmds}$ or $[\![\, C \,]\!] \in \mathit{TCmds}$, to every syntactic command C formed by the grammar *comm* given in Section 2.2. As a particular case one defines $[\![\,\mathsf{skip}\,]\!] =_{df} I$ where I denotes the identity relation defined by $\sigma\, I\, \sigma' \;\Leftrightarrow\; \sigma = \sigma'$ for $\sigma, \sigma' \in \mathit{States} \cup \{\,\mathsf{abort}\,\}$. Moreover, for the remaining commands we additionally specify all required correctness conditions like e.g., all values of free occurring variables being defined within the domains of the stores involved or that only allocated heap cells are accessible. In the Appendix A.3, a definition of the functions $\mathrm{FV}(C)$, the set of free variables and $\mathrm{MV}(C)$, the set of modified variables w.r.t. a command C is provided. In the case of (Boolean) expressions we assume that a corresponding function FV is predefined. The interpretations for the commands that are not heap-dependent can be found in Figure 4.1.

For a more compact notation, we abbreviate the semantic definition by the use of a convention similar to that of the refinement calculus provided e.g. in [BvW98]. A relation R will be defined by a formula F linking input states (s,h) with output ones (s',h'), i.e., we define that $R \mathrel{\widehat{=}} F$ abbreviates the clause $(s,h)\, R\, (s',h') \;\Leftrightarrow_{df}\; F$. As a particular case, it is required for the denotational model that we also assign to Boolean expressions b a relational semantics to manage the definition of the commands involving if - conditionals and while - loops. We view them abstractly as assertion commands $\mathsf{assume}\; b$ that only output the unchanged input state that satisfies

the condition b and otherwise it does not contain such a pair of states as a relation and hence behaves as the empty relation.

$$
\begin{aligned}
[\![v := e]\!] &\mathrel{\widehat{=}} \{v\} \cup \mathrm{FV}(e) \subseteq dom(s) \wedge s' = (v, e^{\mathcal{S}}) \,|\, s \wedge h' = h\,,\\
[\![b]\!] &\mathrel{\widehat{=}} \mathrm{FV}(b) \subseteq dom(s) \wedge b^{\mathcal{S}} = \mathsf{true} \wedge s' = s\,,\\
[\![P\,; Q]\!] &\mathrel{\widehat{=}} \mathrm{FV}(P) \cup \mathrm{FV}(Q) \subseteq dom(s) \wedge (s,h)\, S\, (s',h')\\
&\qquad \text{where } S = [\![P]\!]\,; [\![Q]\!]\,,\\
[\![\mathsf{if}\ b\ \mathsf{then}\ P\ \mathsf{else}\ Q]\!] &\mathrel{\widehat{=}} \mathrm{FV}(b) \cup \mathrm{FV}(P) \cup \mathrm{FV}(Q) \subseteq dom(s) \wedge (s,h)\, S\, (s',h')\\
&\qquad \text{where } S = [\![b]\!]\,; [\![P]\!] \cup \neg [\![b]\!]\,; [\![Q]\!]\,,\\
[\![\mathsf{while}\ b\ \mathsf{do}\ P]\!] &\mathrel{\widehat{=}} \mathrm{FV}(b) \cup \mathrm{FV}(P) \subseteq dom(s) \wedge (s,h)\, S\, (s',h')\\
&\qquad \text{where } S = ([\![b]\!]\,; [\![P]\!])^*\,; \neg [\![b]\!]\,,\\
[\![\mathsf{newvar}\ v\ \mathsf{in}\ P]\!] &\mathrel{\widehat{=}} v \in dom(s) \wedge \exists i \in \mathit{Values} : ((v,i) \,|\, s,h)\, [\![P]\!]\, (s'',h') \wedge\\
&\qquad\qquad s' = (v, s(v)) \,|\, s''\,,\\
[\![\mathsf{newvar}\ v := e\ \mathsf{in}\ P]\!] &\mathrel{\widehat{=}} v \in (dom(s) - \mathrm{FV}(e)) \wedge \mathrm{FV}(e) \subseteq dom(s) \wedge\\
&\qquad\qquad ((v, e^{\mathcal{S}}) \,|\, s,h)\, [\![P]\!]\, (s'',h') \wedge s' = (v, s(v)) \,|\, s''\,.
\end{aligned}
$$

Figure 4.1: Relational semantics of heap-independent commands.

The formulas for assignments to variables and newvar constructs ensure that the expressions involved are defined, i.e., the considered stores contain a definition for all its free variables. The latter construct is rarely used as ambiguities on variables are often resolved by renaming. Sequential composition of commands is translated into relation composition and is defined in the standard way by $\sigma\, R;S\, \sigma' \Leftrightarrow \exists \sigma'' :\ \sigma\, R\, \sigma'' \wedge \sigma''\, S\, \sigma'$ for states $\sigma, \sigma', \sigma''$ and relations R, S. The denotations of the remaining commands are derived from more abstract results which are given in the setting of a Kleene algebra with tests, e.g., [FL79, Koz00]. Conditional statements are defined using $\cup$ which would correspond in programs to non-deterministic choice. For modelling the branch condition assertional commands are applied. The definition for while - loops is similar. As long as b holds the command $[\![b]\!]\,; [\![P]\!]$ is repeated. This coincides denotationally with reflexive transitive closure $(\_)^*$ of the command. After the execution of the loop, the command $\neg [\![b]\!]$ ensures that the final state satisfies the assertion $\neg b$. Note, that the interpretations of the commands given avoid the use of abort and hence can be used for a partial as well as for a total correctness treatment.

Next we continue with relational interpretations of the remaining heap manipulating commands which can produce memory faults. In a total correctness treatment the behaviour of faulting and non-terminating commands is identified. Hence we can

simply use the semantics with the same notation (cf. Figure 4.2). The situation is a bit different for a partial correctness treatment. Note that $\widehat{=}$ only defines non-aborting executions of commands. Hence, for partial correctness it is necessary to additionally define $(s,h)\,[\![\, C \,]\!]\,\mathsf{abort}$ for a state (s,h) and command C iff the condition on the domain of the heap h involved is not satisfied as in the operational semantics of Section 2.2. Moreover, we assume for each such relation R that $\{(\,\mathsf{abort}\,,\,\mathsf{abort}\,)\} \subseteq R$, i.e., the image of abort is abort.

$$
\begin{aligned}
[\![\, v := \mathsf{cons}\,(e_1,...,e_n) \,]\!] \;\widehat{=}\; & \{v\} \cup \mathrm{FV}(e_1) \cup \ldots \cup \mathrm{FV}(e_n) \subseteq \mathit{dom}(s) \wedge \\
& \exists\, a \in \mathit{Addresses} : s' = (v,a) \,|\, s \wedge \\
& a,\ldots,a+n-1 \notin \mathit{dom}(h) \wedge \\
& h' = \{(a,e_1^{\mathcal{S}}),\ldots,(a+n-1,e_n^{\mathcal{S}})\} \,|\, h\,, \\
[\![\, v := [e] \,]\!] \;\widehat{=}\; & \{v\} \cup \mathrm{FV}(e) \subseteq \mathit{dom}(s) \;\wedge\; s' = (v, h(e^{\mathcal{S}})) \,|\, s \;\;\wedge\;\; h' = h\,, \\
& \wedge\; e^{\mathcal{S}} \in \mathit{dom}(h)\,, \\
[\![\, [e_1] := e_2 \,]\!] \;\widehat{=}\; & \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2) \subseteq \mathit{dom}(s) \;\wedge\; s' = s \;\wedge\; h' = (e_1^{\mathcal{S}}, e_2^{\mathcal{S}}) \,|\, h \\
& \wedge\; e_1^{\mathcal{S}} \in \mathit{dom}(h)\,, \\
[\![\, \mathsf{dispose}\; e \,]\!] \;\widehat{=}\; & \mathrm{FV}(e) \subseteq \mathit{dom}(s) \;\wedge\; s' = s \;\wedge\; e^{\mathcal{S}} \in \mathit{dom}(h) \wedge \\
& h' = h - \{(e^{\mathcal{S}}, h(e^{\mathcal{S}}))\}\,.
\end{aligned}
$$

Figure 4.2: Relational semantics of heap-dependent commands.

A similar denotational model for separation logic using a relational approach can also be found in [ORY09] given in a partial correctness setting. The relational interpretations are equivalent to the above ones except for the case of faulting. It is handled there in a different and non-standard way. Before discussing the difference to our approach we first recapitulate that by the use of relations for modelling the programming semantics of separation logic, some care has to be taken for a correct treatment of executions that might abort. The reason is that relation composition generally behaves angelically, i.e., whenever there exists a possibility that an execution can finish successfully such a transition will be taken.

Example 4.1.2 As an example consider the relation $R = \{(\sigma, \mathsf{abort}\,), (\sigma,\sigma')\}$ that allows from a state σ both a successful execution ending in a final state σ' and one that aborts. We further assume for a state $\sigma'' \neq \mathsf{abort}$ a relation $S = \{(\sigma',\sigma'')\}$ that, deviating from our treatment for relational interpretations of partial correctness semantics, ignores aborting executions. Then the composition $R\,;S$ yields the result $\{(\sigma,\sigma'')\}$ ignoring the erroneous execution of R. □

Generally, to obtain validity of the frame rule the converse behaviour is required in separation logic, i.e., considering Example 4.1.2, only the aborting execution of R should remain in the composed relation $R\,;S$. Now a treatment of abort with relational composition leads in [ORY09] to a setting that involves non-standard and more complex definitions for sequential composition of programs and within a proper definition for loops. The relational calculus we use in the following rather takes a different direction. For managing faulting executions we assume additional structure on the commands involved so that relation composition and reflexive transitive closure as well as all general well-known results of these operations can be reused.

Another treatment that deals with the possibility of program faults is provided by the approach of *demonic relational semantics* that can be found e.g. in [Ngu91, BvdW93, DBS+95, DMN97]. As can be imagined from its name that approach uses non-deterministic choice of programs in a demonic fashion, i.e., whenever the program has the possibility of faulting then it will go wrong. A more abstract and algebraic treatment for that structures can be found in [DMT06] in terms of idempotent semirings and Kleene algebras. Semirings are a special case of quantales where no underlying complete lattice is assumed. A total correctness view is taken in that work in the following sense: a state σ only belongs to the domain of a command relation iff no execution starting from σ may lead to an error. Hence, using this semantics it is not needed to include abort into the carrier set of states. But relation composition operators within that approach become in a demonic treatment more complex, since one has to use the demonic variants of the usual (angelic) operators of union and sequential composition.

An alternative would be to use monotonic predicate transformers [Pre09]. However, this would step outside the current relational framework and the aim for a simple algebraic calculus.

We will see in the following that in the provided algebraic treatment a distinguishing of partial and total correctness will not be essential. The reason for this is that in the abstract treatment of commands all involved properties will be given within a so-called *fault-avoiding* behaviour [YO02].

4.2 On Partial and Total Correctness

All subsequent results are formalised abstractly in terms of arbitrary separation algebras. The relationships of the standard model based on the carrier set *States* to (multi-unit) separation algebras denoted by $(\Sigma, \bullet, U)$ where U is a set of units. Their structure is provided at the end of Section 3.3. By this, all abstractions and results that we introduce in the sequel are applicable for the concrete resource model w.r.t.

the carrier set *States* . Since the set of single-unit separation algebras is a strict subset of the set of multi-unit ones, we give all definitions in the following on the latter algebras.

We start with all basic definitions and notations for the used relational structure. A command now essentially is a relation $R \subseteq \Sigma \times \Sigma$, except for the treatment of faults. Clearly, all well-known operations including sequential composition $;$, given by

$$R ; S =_{df} \{(\sigma_1, \sigma_2) : \exists \sigma_3 : (\sigma_1, \sigma_3) \in R \wedge (\sigma_3, \sigma_2) \in S\}$$

and reflexive transitive closure $(\_)^*$ are available, assuming a separation algebra $(\Sigma, \bullet, U)$ with $\sigma_i \in \Sigma$. We introduce a distinct element $\sigma_\perp \notin \Sigma$ that abstractly models program abortion in the relational structure for partial correctness which is given by

$$(\mathcal{P}(\Sigma \times (\Sigma \cup \{\sigma_\perp\}) \cup \{(\sigma_\perp, \sigma_\perp)\}, \subseteq, ;, I)$$

while for total correctness it is given by the simpler structure $(\mathcal{P}(\Sigma \times \Sigma), \subseteq, ;, I)$. Concrete examples of that structures are $(PCmds, \subseteq, ;, I)$ and $(TCmds, \subseteq, ;, I)$ applying the structural transformations defined in Section 3.3. Both abstract structures form Boolean quantales with identity relations I w.r.t. their corresponding carrier set, i.e., $I = \{(\sigma, \sigma) : \sigma \in \Sigma\}$ in the total correctness case and $I = \{(\sigma, \sigma) : \sigma \in \Sigma \cup \{\sigma_\perp\}\}$ in the other case. The greatest element $\top$ of each structure coincides with the universal relation, again w.r.t. the underlying carrier set.

We continue with modelling Hoare triples $\{p\}\, C\, \{q\}$ of separation logic (cf. Definition 2.2.1) using the relation-based denotation. For this we follow a standard algebraic approach given in [Koz00]. There, Hoare logic is algebraically modelled using more abstract structures like semirings or quantales. The role of the pre- and postconditions p and q are played in that approach by tests (cf. Definition 3.2.13), i.e., in the concrete relational quantales elements that coincide with subsets of the identity relation. Note that on tests relation composition $;$ and binary intersection $\cap$ coincide. Tests are also given by relations of the form

$$\widehat{[\![p]\!]} = \{(\sigma, \sigma) \mid \sigma \in [\![p]\!]\} \tag{4.1}$$

for arbitrary set of states $[\![p]\!]$ as defined in Section 3.1. This immediately yields a direct embedding of elements of the assertion quantale AS as tests into a relational quantale. By this we can directly reuse all results of the previous section also for the relational structure. It is clear that the above subidentities and sets of states are in one-to-one correspondence. The set $[\![p]\!]$ can be retrieved from $\widehat{[\![p]\!]}$ by $[\![p]\!] = dom(\widehat{[\![p]\!]})$. We will write $\ulcorner C$ in the following for a command C to denote $\widehat{dom(C)}$, i.e., its domain in a relational subidentity structure. It is characterised pointfree by the universal property (e.g. [DMS06])

$$\ulcorner R \subseteq p \;\Leftrightarrow\; R \subseteq p ; R, \tag{reldom}$$

where p ranges over the set of tests. The locality property $\ulcorner(R\,;\ulcorner S) \subseteq \ulcorner(R\,;S)$, which is independent of the above equivalence, is satisfied for the concrete case of relations. It states that the domain of a composed relation $R\,;S$ depends only on the domain of S, but not on S itself. Symmetrically, one can also define the codomain operation $\vec{R}$ for relations R. In what follows we will write $[\![\, p \,]\!]$ to denote to the corresponding subidentity instead of a set of states. A related approach on relational interpretations for assertions can be found in [TBY12]. Their approach differs to the present one in using n-ary relations with applications in information hiding. For our purposes we do not require such a lift.

As a next step, we turn to a relation-algebraic characterisation of Hoare triples in separation logic beginning with partial correctness behaviour. Note that according to Definition 2.2.1 the semantics of these triples is different from that of standard Hoare logic approaches. An additional assumption is required to guarantee that none of the executions of a considered command aborts from any state satisfying the involved pre-condition. The idea behind this is that the semantics is treated as *resource-sensitive* to obtain proper triples and in particular validity of the frame rule. As an example, one cannot obtain for the triple $\{\,\mathsf{emp}\,\}\, v := [e]\, \{\,?\,\}$ a valid postcondition expressible with the syntax of the assertion language given in Section 2.1. Moreover, this means that the Hoare triples in separation logic are defined to be *fault-avoiding*.

We now continue to extend a prior and well-known algebraic approach for propositional Hoare logic triples with this particular behaviour so that program abortion is also considered. For this we implicitly assume for notational abbreviation an arbitrary translation of syntactical commands and assertions to relations and tests respectively. A denotational semantics of Hoare triples $\{p\}\, C\, \{q\}$ in a partial correctness setting for standard Hoare logic (e.g. [Koz02]) can be given for relations by

$$p\,;C \subseteq C\,;q \;\Leftrightarrow\; p\,;C\,;\neg q \subseteq \emptyset \;\Leftrightarrow\; p\,;C = p\,;C\,;q \tag{4.2}$$

where p, q are suitable subidentities. The formula involving test negation can be difficult to use for our purposes since there we have only general laws for the interplay of concrete assertions of the form $p * q$ and logical negation. The first formula compared to the last one allows inequational reasoning which is often easier to handle. Hence we will mainly use the first version as an adequate formulation in calculations.

For an extension of the approach to incorporate faulting executions of programs we start by defining a special test $\bot =_{df} \{(\sigma_\bot, \sigma_\bot)\}$ as in [DGM+14]. It is used in the sequel for pointfree formulations of conditions involving program abortion.

Definition 4.2.1
A relation C *respects* $\bot$ iff $\bot\,;C = \bot$.

This law states that $\bot$ is a left annihilator on the set of $\bot$-respecting relations. In

a pointwise form this corresponds to relations C satisfying $\sigma_\perp \, C \, \sigma \Rightarrow \sigma = \sigma_\perp$. Note that for the tests p arising in Section 4.1 we have $p \cap \perp = \emptyset$. This allows a relational characterisation of heap-dependent commands. Concrete examples are the mutation, dereferencing and disposal commands of Section 2.2. Clearly, the identity relation also respects $\perp$.

Lemma 4.2.2 *$\sigma_\perp$-respecting relations are closed under $\cup, ;$ and Kleene star *.*

The proof is an easy calculation using the well-known star induction law $R; S \subseteq R \Rightarrow R; S^* \subseteq R$ for any relations R, S (e.g., [Con71]). This guarantees that aborting executions will not be ruled out by union or composition of abort respecting commands. For characterising the above mentioned safety condition that ensures the absence of aborting executions we define that a state σ is *safe* w.r.t. a relation C iff $\neg(\, \sigma \, C \, \sigma_\perp \,)$ holds. In a pointfree fashion, this can be formalised by $\lceil(C \,;\perp) \subseteq \emptyset$ or equivalently $p; \lceil(C;\perp) \subseteq \emptyset$ for arbitrary tests p. Intuitively, p includes at most the states where C will not abort if it starts from them. For a more compact and intuitive representation we now introduce special modal operators that reflect behaviour more concisely (cf. e.g. [MS06a]). We give definitions for our concrete relation-based setting although it is not difficult to lift all subsequent results and handle them more abstractly within modal Kleene algebras.

Definition 4.2.3 (Forward diamond and box)
For an arbitrary relation C and test p we define

$$|C\rangle \, p =_{df} \lceil(C \,; p) \quad \text{and} \quad |C] \, p =_{df} \neg|C\rangle\neg p = \neg\lceil(C \,; \neg p) \,.$$

The forward diamond $|C\rangle \, p$ denotes the test that represents all initial states from which p can be reached within one execution step of C while the forward box $|C] \, p$ dually describes a demonic variant where p must be reached within every single C-step.

With the help of these definitions we can immediately infer

$$p \,; \lceil(C \,; \perp) \subseteq 0 \;\Leftrightarrow\; p \subseteq \neg\lceil(C \,; \perp) \;\Leftrightarrow\; p \subseteq \neg\lceil(C \,; \neg(\neg\perp)) \;\Leftrightarrow\; p \subseteq |C]\neg\perp \,.$$

Hence we can give a pointfree characterisation of the set of safe states using the modal box operator. For better readability we introduce for a test p the abbreviation $\widetilde{p} =_{df} p \,; \neg\perp$. This corresponds to the abort-free part of p. As a particular case we get $\widetilde{I} = \neg\perp$.

Definition 4.2.4
For a relation C we define $\mathsf{safe}\,(C) =_{df} |C] \, \widetilde{I} = \neg\lceil(C \,; \perp)$. By this we characterise all *safe* states of a relation C. We call a test p *safe for* C iff $p \subseteq \mathsf{safe}\,(C)$.

As an immediate consequence of this we give the following lemma.

Lemma 4.2.5 *For any relations C, D we have* $\mathsf{safe}\,(C \cup D) = \mathsf{safe}\,(C) \cap \mathsf{safe}\,(D)$. *Moreover, if D respects $\bot$ then* $\mathsf{safe}\,(C\,;D) = \mathsf{safe}\,(C)\,;\mathsf{safe}\,(C\,;\widetilde{I}\,;D)$. *Hence,* $\mathsf{safe}\,(C\,;D) = \mathsf{safe}\,(C)\,;\,\mathsf{safe}\,(C\,;D)$.

Proof. The first claim follows since box operations are antidisjunctive in the first argument. For the second we calculate

$$\begin{aligned}
\mathsf{safe}\,(C\,;D) &= \neg\lceil(C\,;D\,;\bot)\\
&= \neg(\lceil(C\,;\widetilde{I}\,;D\,;\bot) \cup \lceil(C\,;\bot\,;D\,;\bot))\\
&= \neg\lceil(C\,;\widetilde{I}\,;D\,;\bot) \cap \neg\lceil(C\,;\bot\,;D\,;\bot)\\
&= \mathsf{safe}\,(C\,;\widetilde{I}\,;D) \cap \neg\lceil(C\,;\bot\,;\bot)\\
&= \mathsf{safe}\,(C\,;\widetilde{I}\,;D)\,;\mathsf{safe}\,(C)
\end{aligned}$$

using the definition, distributivity of $\lceil\_$, De Morgan and idempotence of tests, and that ; and $\cap$ coincide on tests. □

As discussed at the end of Section 4.1, this again reflects a demonic treatment of abort respecting relations. Note that the test $|C\rangle\,\widetilde{I}$ as an angelic variant would not be adequate to characterise safe states since it only represents the set of all initial states for which there exists a non-faulting execution of C. To see this, consider Example 4.1.2 involving the relation $C = \{(\sigma, \sigma_\bot), (\sigma, \sigma')\}$. Clearly, we have $(\sigma, \sigma) \in |C\rangle\,\widetilde{I}$, but still σ can lead to program abortion. However, the box operator is exactly what we need in this case. It rules out σ by only including only all initial states of the execution paths of C that can not abort.

Finally, we can now discuss some candidates for a relational characterisation of Hoare triples in separation logic. For the forward box and the backward diamond operation defined by $\langle C|p =_{df} \lceil(p\,;C)\rceil$ there exists a relationship given by the Galois connection

$$p \subseteq |C]\,q \Leftrightarrow \langle C|\,p \subseteq q\,. \tag{4.3}$$

As a particular case, we get $p \subseteq |C]\,\widetilde{I}$ is equivalent to $\langle C|\,p \subseteq \widetilde{I}$. Note that equivalently to (4.2) we can also use $\langle C|\,p \subseteq q$ involving the backward diamond operation for a relational treatment of Hoare triples (e.g. [MS06a]) that are not resource sensitive. Hence, by the characterisation of binary infima we immediately infer $\langle C|\,p \subseteq q \wedge \langle C|\,p \subseteq \widetilde{I} \Leftrightarrow \langle C|\,p \subseteq \widetilde{q}$. This form is still not fully adequate for our purposes due to the asymmetry by just excluding $\bot$ in the postcondition q. One problem with this form is that will particularly falsify validity of the Hoare logic while- inference rule. Thus, we define

Definition 4.2.6 (Partial correctness with abortion)
A *partial correctness* Hoare triple in separation logic is relationally given by

$$\begin{aligned} \{p\}\,C\,\{q\} \;&\Leftrightarrow_{df}\; \langle C|\,\widetilde{p} \subseteq \widetilde{q} \\ &\Leftrightarrow\quad \widetilde{p}\,;C \subseteq C\,;\widetilde{q}\,. \end{aligned}$$

Another possibility for Definition 4.2.6 would be to use $\langle C|\,p \subseteq q \wedge q \subseteq \widetilde{I}$ which implies the above condition and therefore is stronger. We will stay with the above definition since it is more compact and simpler to use. Moreover, it structurally coincides with the original form of relational Hoare triples ignoring program abortion. Therefore we can immediately instantiate existing proofs (e.g. [MS06a]) without any further need for recalculations and state the following result.

Theorem 4.2.7 *All partial correctness inference rules of propositional Hoare logic remain valid under the partial correctness interpretation of Hoare triples respecting program abortion.*

Note that the proof can be lifted to the more abstract setting of a modal Kleene algebras as in [DGM+14]. Hence we can again get (semi)automated and calculational soundness proofs of the Hoare logic proof rules by the use of theorem proving systems. Another advantage of the encoding of the Hoare triples in Definition 4.2.6 is that it also implies that the involved precondition p only characterises safe states.

Lemma 4.2.8 *$\{p\}\,C\,\{q\}$ implies $\widetilde{p}$ is safe for C.*

Proof. By (4.3), isotony of box in its second argument, and definition of $\mathsf{safe}\,(\_)$:

$$\langle C|\,\widetilde{p} \subseteq \widetilde{q} \;\Leftrightarrow\; \widetilde{p} \subseteq |C]\,\widetilde{q} \;\Rightarrow\; \widetilde{p} \subseteq |C]\,\widetilde{I} \;\Leftrightarrow\; \widetilde{p} \subseteq \mathsf{safe}\,(C)\,.$$

□

As a last result we state that the definitions provided in Section 4.1 in fact satisfy Definition 4.2.6 and hence can be used for a relational treatment for Hoare triples of separation logic.

Lemma 4.2.9 *The relational denotations $[\![\,\_\,]\!]$ for commands and assertions in Section 4.1 satisfies the partial correctness interpretation of Hoare triples respecting program abortion, i.e., $\{p\}\,C\,\{q\} \Leftrightarrow [\![p]\!]\,;[\![C]\!] \subseteq [\![C]\!]\,;[\![q]\!]$.*

Proof. By definition of the interpretations $[\![p]\!]$ and $[\![q]\!]$ we can conclude that

$$\widetilde{[\![p]\!]} = [\![p]\!]\,;\neg\bot = [\![p]\!]$$

since $[\![p]\!] \cap \bot \subseteq \emptyset$ and $\cap$ coincides with ; on tests. □

Next we turn to the case of total correctness. We recapitulate the semantics in this approach: A state σ only belongs to the domain of a relation iff there exists an execution starting from σ that does not abort and terminates in some final state τ. Hence, program abortion and non-termination are identified and coincide denotationally with divergence, i.e., the empty relation $\emptyset$. Therefore we need to state for total correctness Hoare triples that the precondition p ensures termination of a command C by assuming relationally $p \subseteq \lceil C$.

Definition 4.2.10 (Total correctness)
We define a *total correctness* Hoare triple in separation logic by

$$\begin{aligned} [p]\,C\,[q] \;&\Leftrightarrow_{df}\; \langle C|\, p \subseteq q \;\wedge\; p \subseteq \lceil C \\ &\Leftrightarrow\quad p\,;C \subseteq C\,;q \;\wedge\; p \subseteq \lceil C\,. \end{aligned}$$

Theorem 4.2.11 *All total correctness inference rules of standard Hoare logic remain valid under the total correctness interpretation of Hoare triples.*

Proof. It is only required to show closure of the termination condition $p \subseteq \lceil C$ for each rule. As an example we prove that condition for the sequential composition rule $\{p\}\,C\,\{r\} \wedge \{r\}\,D\,\{q\} \Rightarrow \{p\}\,C\,;D\,\{q\}$. By p being a test, property of domain, $\{p\}\,C\,\{r\}$ and isotony of domain, $\{r\}\,D\,\{q\}$ and isotony of domain, and locality of domain:

$$p \subseteq \lceil C \Rightarrow p \subseteq p;\lceil C \Leftrightarrow p \subseteq \lceil(p;C) \Rightarrow p \subseteq \lceil(C;r) \Rightarrow p \subseteq \lceil(C;\lceil D) \Leftrightarrow p \subseteq \lceil(C;D)\,.$$

Proofs for the remaining inference rules can easily be calculated and automated within the abstract setting of modal Kleene algebras. □

Note, that the while - inference rule needs an extra termination argument. An adequate condition is expressed relationally by $(b\,;C)^* \subseteq (b\,;C)^*\,;\neg b$ which states that each loop of C that starts in a state for which b holds will eventually end in a state where b is false after finitely many C - steps.

It can be seen that beside the well-known Hoare logic inference rules a pointfree validity proof of the central frame rule of separation logic is still missing. In the subsequent section we provide a relational treatment of that inference rule and give pointfree abstractions of properties for establishing it.

4.3 Abstracting Modularity

The frame rule (cf. Section 2.3) allows verification tasks to be performed locally on the required set of resources of a program. The main advantage is that program

proofs become scalable and easier to read and understand. Conversely it allows the embedding of procedure verifications into larger contexts for obtaining global proofs. This behaviour is established by assuming validity of two crucial properties, called the *frame property* and *safety monotonicity*. We recapitulate that, intuitively, commands that satisfy the former property include program executions that may run on a possibly smaller set of resources. The program can be tracked back to a minimal set of resources that guarantees that it will not abort. Moreover, the latter property states that when at least the required allocated resources are available then the considered command can also be executed from any larger states with additional non-relevant resources without aborting.

The objective of this section is to derive relation-algebraic counterparts for the mentioned conditions. This would allow a simpler and abstract soundness proof of the frame rule in a calculational style. We start with some fundamental definitions that yield a relational variant of the separating conjunction. For this we follow the treatment of [DHM11, DM12a] and introduce an extension of the relational structure from the previous section so that an independent treatment of arbitrary partitions of states is possible. For the purpose of abstraction we give all definitions in the sequel on arbitrary separation algebras capturing all models of Section 3.3. We will use the notation of the more generalised multi-unit separation algebras $(\Sigma, \bullet, U)$ and refer to them in the following just by the term "separation algebra". Moreover, depending on a total or partial correctness treatment we will use Σ or the extended carrier set $\Sigma_\perp =_{df} \Sigma \cup \{\sigma_\perp\}$. For an appropriate treatment of state splittings within $\Sigma_\perp$ we need to extend the separation algebra operation $\bullet$ to also capture $\sigma_\perp$ by

$$\sigma \bullet \tau = \sigma_\perp \Leftrightarrow_{df} \sigma = \sigma_\perp \vee \tau = \sigma_\perp \,, \tag{abortext}$$

i.e., each state denoting program abortion yields again $\sigma_\perp$ in the join. Hence, we always have for any state σ that $\sigma_\perp \# \sigma$.

Definition 4.3.1 (Split and Join)
Assume a separation algebra $(\Sigma, \bullet, U)$ where U denotes a set of units. The *split* relation $\lhd \subseteq \Sigma \times (\Sigma \times \Sigma)$, respectively $\Sigma_\perp \times (\Sigma_\perp \times \Sigma_\perp)$, is given by

$$\sigma \lhd (\sigma_1, \sigma_2) \Leftrightarrow_{df} \sigma_1 \# \sigma_2 \wedge \sigma = \sigma_1 \bullet \sigma_2 \,.$$

The *join* relation $\rhd$ is the converse of split, i.e.,

$$(\sigma_1, \sigma_2) \rhd \sigma \Leftrightarrow_{df} \sigma_1 \# \sigma_2 \wedge \sigma = \sigma_1 \bullet \sigma_2 \,.$$

We introduce a special symmetric symbol for it, rather than writing $\lhd^\smile$ where $\_^\smile$ denotes the converse, to ease reading.

The general idea with this definition is to enable calculations on state partitions by extending the setting from relations between states to state pairs.

Definition 4.3.2 (Pairs of relations)
Union, inclusion and composition of pairs of relations are defined componentwise. The *Cartesian product* $R \times S \subseteq (\Sigma \times \Sigma) \times (\Sigma \times \Sigma)$, respectively $R \times S \subseteq (\Sigma_\perp \times \Sigma_\perp) \times (\Sigma_\perp \times \Sigma_\perp)$, of two appropriate relations $R, S \subseteq \Sigma \times \Sigma$, respectively $R, S \subseteq \Sigma_\perp \times \Sigma_\perp$, is defined by

$$(\sigma_1, \sigma_2)\ (R \times S)\ (\tau_1, \tau_2) \Leftrightarrow_{df} \sigma_1\, R\, \tau_1 \wedge \sigma_2\, S\, \tau_2\,.$$

We assume in the sequel that ; binds tighter than $\cap$ and $\times$ while $\cap$ binds tighter than $\times$. It is clear that $\mathsf{id} =_{df} I \times I$ is the identity of composition on pairs while $\top \times \top$ is the largest pair relation where $\top$ denotes the universal relation.

Definition 4.3.3
Tests in the set of product relations are again sub-identities; as before they are idempotent and commute under ;. The Cartesian product of tests is a test again. However, there are other tests, such as the *combinability check* $\# \subseteq (\Sigma \times \Sigma) \times (\Sigma \times \Sigma)$, respectively $(\Sigma_\perp \times \Sigma_\perp) \times (\Sigma_\perp \times \Sigma_\perp)$, on pairs of states, given by:

$$(\sigma_1, \sigma_2)\ \#\ (\tau_1, \tau_2) \Leftrightarrow_{df} \sigma_1 \# \sigma_2 \wedge \sigma_1 = \tau_1 \wedge \sigma_2 = \tau_2\,.$$

This relation acts as a filter since only combinable pairs of states pass it while all other pairs are not considered.

Lemma 4.3.4 *We have* $\# = \rhd ; \lhd \cap \mathsf{id}$ *and hence* $\# \subseteq \rhd ; \lhd$. *Moreover* $\# ; \rhd = \rhd$ *and symmetrically* $\lhd ; \# = \lhd$. *Finally,* $\lhd ; \rhd = I$.

It is well known that $\times$ and ; satisfy an equational exchange law:

$$(R_1 \times R_2)\,;\,(S_1 \times S_2) \;=\; (R_1\,;\,S_1) \times (R_2\,;\,S_2)\,. \qquad (\times/;)$$

Next we lift the operation $\bullet$ and the relation $\#$ on states to the setting of relations.

Definition 4.3.5
The $*$ - *composition* $R * S$ of relations $R, S \subseteq \Sigma \times \Sigma$, respectively $\Sigma_\perp \times \Sigma_\perp$, is defined by the formula

$$R * S =_{df} \lhd\,;\,(R \times S)\,;\,\rhd\,.$$

Intuitively by this definition, the relation $\sigma\ (R * S)\ \tau$ holds iff σ can be split as $\sigma = \sigma_1 \bullet \sigma_2$ with combinable partitions σ_1, σ_2 on which R and S can act and produce

results τ_1, τ_2 that are again combinable and yield $\tau = \tau_1 \bullet \tau_2$. For a concrete example consider the separation algebra $(\mathit{States}, \uplus, \mathsf{emp})$. We can interpret the semantics of the split relation w.r.t. that carrier set as follows: a state (s,h) can be split into states (s,h_1) and (s,h_2) with $h = h_1 \uplus h_2$ iff $\mathit{dom}(h_1) \cap \mathit{dom}(h_2) = \emptyset$. Clearly, we can also embed the operation $\uplus$ into a lifted version on tests, for that we also write $\uplus$, by $\widehat{P} \uplus \widehat{Q} =_{df} \widehat{P \uplus Q}$ for $P, Q \subseteq \mathit{States}$ and $\widehat{\ }$ as defined in (4.1). Assuming the separation algebra *States*, the product $p \uplus q$ of tests p, q would coincide with $\lhd ; (p \times q) ; \rhd$.

In particular, we remark that Definition 4.3.5 reflects angelic behaviour in the sense that, whenever σ_1 and σ_2 are not combinable or disjoint, R and S are prevented from starting, since these states are eliminated by the definition. The same happens if σ_1 and σ_2 are combinable but R and S produce non-combinable output states τ_1 and τ_2.

We will see in the following that the relations of the form $R * S$ can be either used to characterise the structure and behaviour of programs or can be conceptually interpreted as a parallel execution of programs within a concurrent setting. The former case will be discussed in the subsequent section. For now we continue by some consequences of Definition 4.3.5.

First, by definition of separation algebras the partial operator $\bullet$ is associative and commutative. Hence, the lifted operation $*$ is also associative and commutative. Moreover, it has the test $e =_{df} \{(u,u) : u \in U\}$ as its unit. For tests p, q the $*$-composition $p * q$ is a test again, irrespective of the underlying separation algebra.

Lemma 4.3.6 *I is idempotent w.r.t. $*$, i.e., $I * I = I$.*

Proof. We calculate, using the definitions and Lemma 4.3.4,

$$I * I = \lhd ; (I \times I) ; \rhd = \lhd ; \mathsf{id} ; \rhd = \lhd ; \rhd = I.$$

□

For a partial correctness treatment we can immediately infer the following lemma from (abortext).

Lemma 4.3.7

(a) $\neg\bot ; \lhd = \lhd ; (\neg\bot \times \neg\bot)$,

(b) $\bot ; \lhd = \lhd ; (\bot \times I) \cup \lhd ; (I \times \bot)$,

(c) $\bot = C * \bot$ *for arbitrary relations C.*

The proof can be found in Appendix A. Clearly, symmetric laws hold for the converse relation $\rhd$. We can now calculate useful laws that provide characteristics of the interplay between $\bot$ and $*$-products.

Lemma 4.3.8 *For arbitrary tests p, q we have $\widetilde{p * q} = \widetilde{p} * \widetilde{q}$.*

Proof. By definition, Lemma 4.3.7(a) and exchange ($\times$/;), we calculate
$\neg\bot ; (p * q) = \neg\bot ; \lhd ; (p \times q) ; \rhd = \lhd ; (\neg\bot ; p \times \neg\bot ; q) ; \rhd = (\neg\bot ; p) * (\neg\bot ; q)$. □
This means that whenever a $*$-composition of two tests p, q is free of the state denoting program abortion then already p and q themselves do not involve that state and vice versa.

Lemma 4.3.9 *$\bot$-respecting relations are closed under $*$.*

Proof. Assume C, D are $\bot$-respecting relations. We need to show $\bot ; (C * D) = \bot$. The claim follows from the relational definition of $*$, Lemma 4.3.7(b), distributivity, exchange ($\times$/;), the assumption and Lemma 4.3.7(c). □

By interpreting $*$ as a parallel composition of programs the composed program will respect abortion if its constituent programs do. We will elaborate on this in the next section.

Finally, there is the following interplay between $*$ and the domain operator.

Lemma 4.3.10 *For relations R, S we have $\lceil (R * S) \subseteq \lceil R * \lceil S$.*

The proof can be found in the Appendix. Generally the reverse inclusion is not valid since in the right-hand side of the inequation there can exist executions of R and S that come with combinable starting states but also involve final states that are not combinable as required for the relation $R * S$. We will later provide a condition that guarantees an equational relationship.

Finally, we state for readers familiar with *fork algebras* (e.g. [FBH97]) that there exists some relationship to the provided split and join relations. Generally, fork algebras allow more expressibility than standard relation algebras by extending them with a *fork* operation given by

$$R \underline{\nabla} S =_{df} \{(x, \star(y, z)) : xRy \wedge xSz\},$$

where $\star$ denotes an abstract and injective binary *pairing function*. In our concrete case this is given by $\star(\sigma, \tau) = (\sigma, \tau)$ for states σ, τ. An approximation of the join and split relations can be given by $((\succeq) \underline{\nabla} (\succeq)) ; \# = \{(\sigma, (\sigma_1, \sigma_2)) : \sigma \succeq \sigma_1, \sigma \succeq \sigma_2, \sigma_1 \# \sigma_2\}$, where $\succeq$ denotes the converse of $\preceq$. This relation only models a superset of $\lhd$ since

it does not relate combinable substates $\sigma_1, \sigma_2 \preceq \sigma$ in the sense that $\sigma_1 \bullet \sigma_2 = \sigma$ as in the case of $\lhd$. Moreover, the Cartesian products of Definition 4.3.2 coincides with direct products of fork algebras, i.e., $P \times Q = P \otimes Q$.

As a further related approach we mention that [GLW06] also considers splitting and joining of states by a ternary relation within the notion of a relational frame for providing an alternative semantic foundation for Boolean BI. However, for our applications in the following these relations are rather used for characterising behaviours of programs in separation logic.

4.3.1 A Pointfree Frame Property

We are now in the position to derive a relational and pointfree characterisation of the frame property (cf. Section 2.3). First, we start by briefly recapitulating the frame property. It expresses that any execution of a command C can be tracked back to a possibly smaller heap portion that is sufficient for a non-aborting execution. This reads formally for heaps h_0, h_1 with $dom(h_0) \cap dom(h_1) = \emptyset$ as follows

$$\begin{aligned}\neg(\langle C, (s, h_0)\rangle \rightsquigarrow^* \ \mathsf{abort}\,) \ \wedge \ \langle C, (s, h_0 \cup h_1)\rangle \rightsquigarrow^* (s', h') \ \Rightarrow \\ \exists\, h_0' : \langle C, (s, h_0)\rangle \rightsquigarrow^* (s', h_0') \ \wedge \ h' = h_0' \cup h_1 \ \wedge \ dom(h_0') \cap dom(h_1) = \emptyset\,.\end{aligned}$$

The first conjunct of the premise asserts that the command C will not abort starting from the state (s, h_0), i.e., it is a safe state for C. For a pointfree form of this part we can use Definition 4.2.4 as an adequate candidate for a partial correctness treatment and $\lceil C$ for the case of total correctness. The second assumption states that any execution on a larger heap $h_0 \cup h_1$ ending in a final state (s', h') can be resolved as follows: h_0 is modified by C and results in a subheap $h_0' \subseteq h'$ while h_1 represents that part that is not modified and left untouched by C, i.e., the frame of C.

At this point we can use the extended relational structure involving pairs of relations for a pointfree characterisation of that behaviour. The idea is to interpret the configuration $\langle C, (s, h_0 \cup h_1)\rangle \rightsquigarrow^* (s', h')$ relationally by $(s, h_0 \cup h_1)\, C\, (s', h')$ and further rewrite this execution into $(s, h_0)\, C\, (s', h_0')$ as in the conclusion and (s, h_1) ? (s', h_1) where we unfortunately cannot use skip instead of the place holder ? since store variables may be modified by C as $s = s'$ does not generally hold. This means that the behaviour of assignments to store variables is different from the case of heap cells, since the effects become globally visible. Hence, care has to be taken when reassembling the overall final state from the computation on the smaller portion. We can model changes on the store component by a special relation that we introduce as follows.

Definition 4.3.11
Let H be a relation that preserves all heaps while being liberal about the involved stores:

$$(s,h)\,H\,(s',h') \;\Leftrightarrow_{df}\; h = h'\,.$$

It can be seen that H is reflexive, transitive and symmetric, i.e., an equivalence relation. For abstracting this approach we will later use parts of these properties to give a more general characterisation. For now we can use the relation H to formulate a pointfree version of the frame property for the carrier set of *States*.

Definition 4.3.12 (Relational frame property)
A relation C satisfies the *relational frame property* in a total correctness setting iff

$$(\lceil C \times I)\,;\rhd\,;\,C \;\subseteq\; (C \times H)\,;\rhd\,.$$

The partial correctness version can be obtained by replacing $\lceil C$ in the above inequation with $\mathsf{safe}\,(C)$ and I with $\neg\bot$, i.e.,

$$(\mathsf{safe}\,(C) \times \neg\bot)\,;\rhd\,;\,C \;\subseteq\; (C \times H)\,;\rhd\,.$$

Note that we have $(s,h)\;I\;(s,h)$ and respectively $(s,h)\;\neg\bot\;(s,h)$ for any state (s,h) since both relations are subidentities. Now prefixing the join operator with $\lceil C \times I$ or $\mathsf{safe}\,(C) \times \neg\bot$ on the left-hand side asserts that the initial state can be split into two substates with disjoint heaps. In particular, $\lceil C$ and $\mathsf{safe}\,(C)$ denotes the partitions on which C can be safely executed while I and $\neg\bot$ represents the unchanged remaining resources. As a check of adequacy, we further provide a pointwise form of the above inequation for total correctness. The partial correctness frame property can be given analogously. For arbitrary $s, s' \in \mathit{States}$ and $h_0, h_1, h' \in \mathit{Heaps}$ we have

$$\begin{aligned}
&(s,h_0)\;\lceil C\;(s,h_0) \;\wedge\; \mathit{dom}(h_0) \cap \mathit{dom}(h_1) = \emptyset \;\wedge\; (s, h_0 \cup h_1)\;C\;(s',h') \\
\Rightarrow\;& \exists\, h_0', h_1' :\; \mathit{dom}(h_0') \cap \mathit{dom}(h_1') = \emptyset \;\wedge\; h' = h_0' \cup h_1' \\
&\qquad\qquad \wedge\; (s,h_1)\;H\;(s',h_1') \;\wedge\; (s,h_0)\;C\;(s',h_0')\,.
\end{aligned}$$

By the definition of H the conclusion simplifies to the equivalent condition

$$\exists\, h_0' :\; \mathit{dom}(h_0') \cap \mathit{dom}(h_1) = \emptyset \;\wedge\; h' = h_0' \cup h_1 \;\wedge\; (s,h_0)\;C\;(s',h_0')\,.$$

This corresponds to the conclusion given in the definition of the frame property, in a relational fashion.

As mentioned, the derivation of the frame property relies on the concrete relation H which is defined on the states of the carrier set *States* and thus makes the treatment less general. Therefore as a next step we give an abstraction from the relation H to more general relations satisfying suitable properties that ensure the required behaviour [DHM11].

Definition 4.3.13 Given a separation algebra $(\Sigma, \bullet, U)$, a relation $K \subseteq \Sigma \times \Sigma$, respectively $\Sigma_\perp \times \Sigma_\perp$, is called a *compensator* iff it satisfies the following properties:

(a) $I \subseteq K$,

(b) $K ; K \subseteq K$,

(c) $\# ; (I \times K) ; \# \subseteq \#$,

(d) $\neg\perp ; K \subseteq K ; \neg\perp$, in the case of partial correctness.

The requirements (a) and (b) together denote that every compensator is a preorder. By this, arbitrarily long sequences of relational commands can be "accompanied" by equally long compensator sequences. Requirement (c) can be explained by interpreting K as an environment that restricts the allowed behaviour of a relation or more concretely of a command that is acting on combinable parts of a considered state. Now the meaning of (c) is that it will not restrict a command that does not modify anything, i.e., it acts as the identity relation on its combinable part of the state. The last requirement states that compensators does not produce memory or program faults. Note that for a total correctness treatment, one can replace $\perp$ with $\emptyset$ in that inequation which makes it trivially satisfied. These requirements are used amongst others to establish closure of properties involving compensators, in particular the frame property.

There exists a relationship of compensators to so-called *interference relations* provided within a concurrent context in [DYBG+13]. That relations also provide a characterisation of the environment that restrict allowed behaviour e.g., of threads running in parallel. Concrete details for this approach remain as future work. We continue by stating some immediate consequences.

Lemma 4.3.14 *The relations H and I are compensators.*

The proof follows immediately from the definitions.

Lemma 4.3.15 *If K_1, K_2 are compensators then also $K_1 \cap K_2$ is a compensator.*

Proof. For requirement (a) of Definition 4.3.13 it is clear while (b) follows immediately from isotony. The third property is calculated by (sub)distributivity and assumptions:

$$\begin{aligned} \# ; (I \times K_1 \cap K_2) ; \# &= \# ; ((I \times K_1) \cap (I \times K_2)) ; \# \\ &\subseteq \# ; (I \times K_1) ; \# \cap \# ; (I \times K_2) ; \# \\ &\subseteq \# \cap \# \\ &= \# . \end{aligned}$$

Finally, for a test p we always have that $p\,;(R \cap S) = p\,;R \cap p\,;S$, (e.g. [Möl07]). Hence, we can calculate

$$\neg\bot\,;(K_1 \cap K_2) = (\neg\bot\,;K_1) \cap (\neg\bot\,;K_2) \subseteq (K_1\,;\neg\bot) \cap (K_2\,;\neg\bot) = (K_1 \cap K_2)\,;\neg\bot\,.$$

□

Finally, we present a generalised frame rule that is parametric w.r.t. partial or total correctness and the considered compensation relation K.

Definition 4.3.16 (Generalised frame property)
Assume a compensator K. Then a relation C has the *generalised frame property* for total correctness iff

$$(\lceil C \times I)\,;\rhd\,;\,C \subseteq (C \times K)\,;\rhd$$

and in the case of partial correctness iff

$$(\mathsf{safe}\,(C) \times \neg\bot)\,;\rhd\,;\,C \subseteq (C \times K)\,;\rhd\,.$$

Note that both inequations are equivalent to respective formulas with $\#\,;(C \times K)\,;\rhd$ as the right-hand side. In particular, it can be seen that the frame property has for both treatments the same relational structure which allow soundness proofs in the following in a largely unified fashion.

Lemma 4.3.17 *I has the frame property. In the case of total correctness all tests have the frame property.*

Proof. By definitions, $(s \times I)\,;\rhd\,;\,p = (I \times I)\,;\rhd \subseteq (I \times K)\,;\rhd$ in the case of total correctness. Moreover, the other case follows from $\mathsf{safe}\,(I) = \neg\bot \subseteq I$. □

Since more complex commands are built up from simpler ones using the $\cup$ and $;$ operators, we further show that, subject to suitable conditions, the frame property is closed under them.

Lemma 4.3.18 *Assume a compensator K. The generalised frame property is closed under union, composition. For partial correctness the frame property propagates from C and D only to $C\,;D$ if additionally D respects $\bot$.*

The proof can be found in Appendix A. In particular, it can be seen that the proofs for partial and total correctness are nearly the same. Both tests $\lceil\_$ and $\mathsf{safe}\,(\_)$ satisfy the required laws that are used for a unified proof of these closure conditions. The additional assumption that the composed command D respects $\bot$ is required since we use the standard definition of relational composition. All executions of C that produce

a program fault need to be maintained in $C\,;D$. Since relation composition behaves angelic we therefore assume $\bot$-respecting relations. The relational denotation provided in [ORY09] uses, compared to our approach, a different definition of relational composition that exactly takes these effects into account. In the total correctness treatment this is not required since only complete and non-faulting runs of programs are considered like in the formal approach to separation logic given in [YO02].

Corollary 4.3.19 *The frame property for total correctness propagates for a compensator K from relations C and D to* $\mathsf{if}\ b\ \mathsf{then}\ C\ \mathsf{else}\ D$ *for arbitrary tests b.*

Unfortunately, for a partial correctness treatment an analogue result cannot be obtained. The reason for this is that I and $\emptyset$ are the only tests that satisfy the frame property. However, one still obtains a (stronger) variant that can be applied for the corresponding inference rules. Assume a relation C satisfies the frame property then for arbitrary tests b

$$(b\,;\mathsf{safe}\,(C)\ \times\neg\bot)\,;\rhd\,;b\,;C\ \subseteq\ (b\,;C\ \times K)\,;\rhd\,.$$

4.3.2 Resource Preservation

A further ingredient is still needed for a pointfree soundness proof of the frame rule. Note that the frame rule comes with a side condition on the variables involved, i.e., $\mathrm{FV}(r)\cap\mathrm{MV}(C)=\emptyset$. It ensures that the variables modified by the command C are distinct from the free variables of the untouched part r. It remains to express algebraically the requirement that a command preserves certain variables. For this we can avoid an explicit mentioning of syntax and free variables by finding a suitable purely algebraic condition instead. The main idea of that property is to relationally express that a test r, which represents the untouched part within the frame rule, will not be changed by a considered compensator K within $\times$-products of relations.

Definition 4.3.20
A relation C *preserves* a test r w.r.t. a compensator K iff

$$\lhd\,;(C\times\ r\,;K)\,;\#\ \subseteq\ \top\,;\lhd\,;\,(I\times r)\,.$$

The informal explanation is as follows: when C is executed on parts of the state distinct from combinable part that satisfies r then every re-assembled final state must contain an r-part, too. Equivalently, we can replace the right-hand side by $\top\,;\lhd\,;(I\times r)\,;\#$ which states the final parts have to be $\#$-combinable. Moreover the definition of preservation is also equivalent to

$$\#\,;(C\times\ r\,;K)\,;\#\ \subseteq\ \#\,;(\top\times\ \top\,;r)\,;\#\,. \tag{4.4}$$

A proof of this is deferred to Appendix A. This characterisation does not use the $\rhd$- and $\lhd$-relations and will be simpler to use for a pointwise derivation of properties. Other variants that allow similar characterisations of preservation can be found in [DH11]. Next we show that our notion of preservation fits well with the original side condition of the frame rule. It is trivially satisfied for all assertions r if $\mathrm{MV}(C) = \emptyset$. Concrete commands with this property are in separation logic, e.g., mutation commands $[e_1] := e_2$ and dispose. We show that this condition implies our notion of preservation, so that it is adequate, but also more liberal than the original one.

Lemma 4.3.21 *Consider the compensator H and the carrier set States. If $\mathrm{MV}(C) = \emptyset$ for a command C then its relational denotation preserves all tests r.*

Proof. The assumption implies that C cannot change the store part of any state, but only heap parts. For states $\sigma, \tau \neq$ abort we formally have

$$\forall\, \sigma, \tau :\ \sigma\ C\ \tau\ \Rightarrow\ s_\sigma = s_\tau\,, \tag{4.5}$$

where s_σ denotes the store of the state σ. By applying the pointwise definitions, r is a subidentity, $\tau_1 \# \tau_2 \wedge \sigma_1 \# \sigma_2$ and assumption (4.5) imply $s_{\sigma_2} = s_{\tau_2}$, the definition of H and $s_{\sigma_2} = s_{\tau_2}$ further imply $\sigma_2 = \tau_2$, logic step and omitting conjuncts, definition of $\top$, and again pointwise definitions, we infer:

$$\begin{array}{ll}
 & (\sigma_1, \sigma_2)\ \#\,;(C \times r\,;H)\,;\#\ (\tau_1, \tau_2) \\
\Leftrightarrow & \sigma_1 \# \sigma_2 \wedge \sigma_1\ C\ \tau_1 \wedge \sigma_2\ \ r\,;H\ \tau_2 \wedge \tau_1 \# \tau_2 \\
\Leftrightarrow & \sigma_1 \# \sigma_2 \wedge \sigma_1\ C\ \tau_1 \wedge \sigma_2\ r\ \sigma_2 \wedge \sigma_2\ H\ \tau_2 \wedge \tau_1 \# \tau_2 \\
\Leftrightarrow & \sigma_1 \# \sigma_2 \wedge \sigma_1\ C\ \tau_1 \wedge \sigma_2\ r\ \sigma_2 \wedge \sigma_2\ H\ \tau_2 \wedge s_{\sigma_2} = s_{\tau_2} \wedge \tau_1 \# \tau_2 \\
\Rightarrow & \sigma_1 \# \sigma_2 \wedge \sigma_1\ C\ \tau_1 \wedge \sigma_2\ r\ \sigma_2 \wedge \sigma_2 = \tau_2 \wedge \tau_1 \# \tau_2 \\
\Rightarrow & \sigma_1 \# \sigma_2 \wedge \tau_2\ r\ \tau_2 \wedge \tau_1 \# \tau_2 \\
\Leftrightarrow & \sigma_1 \# \sigma_2 \wedge \sigma_1\ \top\ \tau_1 \wedge \sigma_2\ \top\ \tau_2 \wedge \tau_2\ r\ \tau_2 \wedge \tau_1 \# \tau_2 \\
\Leftrightarrow & (\sigma_1, \sigma_2)\ \#\,;(\top \times \top\,;r)\,;\#\ (\tau_1, \tau_2)\,.
\end{array}$$

□

A similar treatment of the general condition $\mathrm{MV}(C) \cap \mathrm{FV}(r) = \emptyset$ for arbitrary C is possible and can be obtained analogously. We continue with some direct consequences of the definition of preservation.

Lemma 4.3.22 *Assume a compensator K. I preserves every test r and it is preserved by any relation.*

Proof. By neutrality of I and exchange ($\times$/;), $\#$ and $I \times r$ are tests, definition of compensators (Def. 4.3.13(b)), isotony,

$$
\begin{array}{ll}
 & \#\,;(I \times r\,;K)\,;\# \\
= & \#\,;(I \times r)\,;(I \times K)\,;\# \\
= & \#\,;(I \times r)\,;\#\,;(I \times K)\,;\# \\
\subseteq & \#\,;(I \times r)\,;\# \\
\subseteq & \#\,;(\top \times \top\,;r)\,;\#\ .
\end{array}
$$

Every command preserves I follows immediately from isotony and neutrality of I. □

Lemma 4.3.23 *Preservation of a test r w.r.t. a compensator K is closed under union and pre-composition with a test. If*

$$\#\,;(C\,;D\ \times K)\,;\#\ \subseteq\ \#\,;(C \times K)\,;\#\,;(D \times K)\,;\# \tag{4.6}$$

for relations C, D then preservation of r propagates from C and D to $C\,;D$.

The proof can be found in Appendix A. Equation (4.6) means that the "local" intermediate state of the composition $C\,;D$ induces a "global" intermediate state; it means a "modular" way of composition. Consider e.g., the triple $\{p * r\}\,C\,;D\,\{s * r\}$ which can be shown if one can infer $\{p * r\}\,C\,\{q * r\}$ and $\{q * r\}\,D\,\{s * r\}$. The existence of such a q is ensured by the insertion of an intermediate $\#$ relation in (4.6).

Corollary 4.3.24 *Preservation w.r.t. a compensator K propagates from relations C and D to* if b then C else D *for arbitrary tests b.*

4.3.3 A Calculational Proof of the Frame Rule

All basic assumptions for providing a validity proof of the frame rule have now been established. In the following we give a completely algebraic soundness proof of a generalised form of the frame rule abstracted to arbitrary separation algebras for a partial and total correctness setting. We start with some consequences for the state splitting relations that in particular will allow a simpler characterisation of Hoare triples using the universal relation $\top$ and thus a more concise and intuitive proof of the frame rule.

Lemma 4.3.25

(a) $\top\,;\lhd = \lhd\,;(\top \times \top)\,;\#$.

(b) *For arbitrary tests p, q we have* $\top\,;\lhd\,;(p \times q)\,;\#\ =\ \lhd\,;(\top\,;p\ \times\ \top\,;q)\,;\#$. *Therefore,* $\top\,;(p * q) = (\top\,;p) * (\top\,;q)$.

Proof.

(a) First,

$$\begin{aligned}\sigma\ (\top\,;\lhd)\ (\rho_1,\rho_2) &\Leftrightarrow \exists\,\rho : \sigma\,\top\,\rho \wedge \rho_1 \# \rho_2 \wedge \rho = \rho_1 \bullet \rho_2\\ &\Leftrightarrow \rho_1 \# \rho_2 \wedge \exists\,\rho : \rho = \rho_1 \bullet \rho_2\\ &\Leftrightarrow \rho_1 \# \rho_2\,.\end{aligned}$$

Second,

$$\begin{aligned}&\sigma\ (\lhd\,;(\top\times\top)\,;\#)\ (\rho_1,\rho_2)\\ \Leftrightarrow\ &\exists\,\sigma_1,\sigma_2 : \sigma_1 \# \sigma_2 \wedge \sigma = \sigma_1 \bullet \sigma_2 \wedge \sigma_1\,\top\,\rho_1 \wedge \sigma_2\,\top\,\rho_2 \wedge \rho_1 \# \rho_2\\ \Leftrightarrow\ &\exists\,\sigma_1,\sigma_2 : \sigma_1 \# \sigma_2 \wedge \sigma = \sigma_1 \bullet \sigma_2 \wedge \rho_1 \# \rho_2\\ \Leftrightarrow\ &\rho_1 \# \rho_2\,,\end{aligned}$$

since we can choose $\sigma_1 = \sigma$ and $\sigma_2 = u$.

(b) Straightforward from Part (a), exchange $(\times/;)$ and the definition of $*$ on relations. □

Hence by the use of the universal relation $\top$ we can now give the following useful equivalent formulations for executions of Hoare triples.

Lemma 4.3.26 *For arbitrary tests p, q and relation C we have*

$$p\,;C \subseteq C\,;q \Leftrightarrow \top\,;p\,;C \subseteq \top\,;q \Leftrightarrow p\,;C \subseteq \top\,;q\,.$$

The proof can be found in Appendix A. The right-hand sides of the inequations are more liberal than in the original formulation. In particular this form is appropriate for our purposes comparing it with the structure of the preservation property that also involves the universal relation in its right-hand side. Next, we derive some further consequences from the definitions that will be applied in a pointfree proof of the frame rule.

Lemma 4.3.27 *Assume a compensator K.*

(a) Suppose a relation C has the generalised frame property. Then for all tests p, r we have

$$\begin{aligned}p \subseteq \ulcorner C &\Rightarrow (p * r)\,;C \subseteq (p\,;C) * (r\,;K)\,,\\ p \subseteq \mathsf{safe}\,(C) &\Rightarrow (p * \widetilde{r})\,;C \subseteq (p\,;C) * (\widetilde{r}\,;K)\,.\end{aligned}$$

The former applies to total correctness while the latter to partial correctness.

(b) Suppose relation C preserves a test r. Then for all tests q we have

$$(C\,;q) * (r\,;K) \subseteq \top\,;(q * r)\,.$$

The proof can be found in Appendix A. Note again that the proof for total and partial correctness can be given in a largely unified fashion parametric w.r.t. the set of safe states and the domain of a considered relation. The generalised frame rule now reads as follows.

Theorem 4.3.28 (Generalised Frame Rule) *Assume a compensator K, a test r and a relation C that has the frame property. If C preserves $\widetilde{r}$ then the partial correctness frame rule is valid:*

$$\frac{\{p\}\,C\,\{q\}}{\{p * r\}\,C\,\{q * r\}}\,.$$

If C preserves r and additionally $\lceil C * I \subseteq \lceil C$ then the total correctness frame rule holds:

$$\frac{[p]\,C\,[q]}{[p * r]\,C\,[q * r]}\,.$$

Proof. First, Lemma 4.2.8 and $\{p\}\,C\,\{q\}$ imply $\widetilde{p} \subseteq \mathsf{safe}(C)$. By Lemma 4.3.8, Lemma 4.3.27(a), isotony and assumption, i.e., $p\,;C \subseteq C\,;q$, Lemma 4.3.27(b), and Lemma 4.3.8:

$$\widetilde{p * r}\,;C \;=\; (\widetilde{p} * \widetilde{r})\,;C \;\subseteq\; (\widetilde{p}\,;C) * (\widetilde{r}\,;K) \;\subseteq\; (C\,;\widetilde{q}) * (\widetilde{r}\,;K) \;\subseteq\; \top\,;(\widetilde{q} * \widetilde{r}) \;\subseteq\; \top\,;\widetilde{q * r}\,.$$

Second, we have by the assumption $p \subseteq \lceil C$ and isotony that $p * r \subseteq \lceil C * I \subseteq \lceil C$. Moreover, we infer by Lemma 4.3.27(a), the assumption and Lemma 4.3.27(b) that

$$(p * r)\,;C \;\subseteq\; (p\,;C) * (r\,;K) \;\subseteq\; (C\,;q) * (r\,;K) \;\subseteq\; \top\,;(q * r)\,.$$

□

One difference of both proofs is that we need to apply Lemma 4.3.8 twice in the case of partial correctness to guarantee $\bot$-freeness on the involved assertions. Moreover, the proof of the frame rule in the total correctness case unfortunately requires the additional assumption $\lceil C * I \subseteq \lceil C$ as a point-free variant of a property called *termination monotonicity* in the common literature [YO02]. Intuitively, if C terminates starting from a state σ it also will terminate from any possibly larger initial state $\sigma \bullet \tau$ assuming $\sigma \# \tau$. The partial correctness case differs in its algebraic treatment as it does not require an additional pointfree variant of *safety monotonicity* property (cf. Section 2.3). The reason for this is that the required part of this property is implicitly incorporated in the relational variants of the Hoare triples in combination with the frame and preservation properties. Their application in the above proofs guarantee that all considered executions of the conclusion will not abort, i.e., end in the final state $\sigma_\bot$. There exist further approaches using state and predicate transformers as a semantic approach to separation logic [YO02, COY07, HHM+11]. These approaches

also come with simple formulations that involve a built-in safety monotonicity condition.

Note that in the above rules the preservation property on the test r denotes the relational counterpart of the side condition stating that the modified variables of C and the free variables of r have to be distinct. This required a modelling of the side effects on r by introducing the notion of a compensator. For the purpose of simplification, related approaches in the literature as e.g., [COY07, HHM+11] does not require and use an appropriate substitute for that. The main reason for this is that the resource models that are considered for those treatments are extended by a permission framework (cf. Example 3.3.3) that handles the variable conditions within the semantics of logic itself rather than using syntactical conditions. The concrete approach to this can be found in [BCY06] and for the case of Hoare logics in [PBC06]. The idea by this is to prevent a command C from modifying a set of variables by restricting its behaviour in the sense that at most a read permission for those variables is held by C so that it can only perform a read access. Since our abstract and general treatment also includes separation algebras that involve permissions we can simplify our relational framework, too. For this, note that by Lemma 4.3.14 I is a compensator. Moreover, by Equation (4.4) we can calculate using neutrality and isotony that

$$\#\,;(C \times r\,;I)\,;\# \;=\; \#\,;(C \times r)\,;\# \;\subseteq\; \#\,;(\top \times \top\,;r)\,;\# \;.$$

Therefore, using the identity relation as a compensator the preservation condition is always satisfied. Hence we can also conclude with a simpler version of the frame rule.

Corollary 4.3.29 (Frame Rule on Permission Algebras) *Assume a test r and a relation C that has the frame property w.r.t. the compensator I. Then the partial correctness frame rule is valid:*

$$\frac{\{p\}\,C\,\{q\}}{\{p*r\}\,C\,\{q*r\}}\;.$$

If additionally $\lceil C*I \subseteq \lceil C$ then the total correctness frame rule holds:

$$\frac{[\,p\,]\,C\,[\,q\,]}{[\,p*r\,]\,C\,[\,q*r\,]}\;.$$

4.3.4 Related Algebraic Approaches

We conclude this section by a discussion on related approaches that use non-relational settings to formalise the framing behaviour of separation logic. The first treatment [COY07] involves the usage of so-called *local actions* defined on the concept

of separation algebras. Basically, local actions are special state transformers, i.e., particular functions that map states to sets of states or to a distinguished element $\top$[1]. The element $\top$ is used to denote program abortion, e.g., due to dereferencing of non-allocated resources.

There is also an order $\sqsubseteq$ defined on sets of states and $\top$. The special case for arbitrary sets of states $p, q \in \mathcal{P}(\Sigma)$ excluding $\top$ is defined by $p \sqsubseteq q =_{df} p \subseteq q$. Moreover, for arbitrary $p \in \mathcal{P}(\Sigma) \cup \{\top\}$ one always has $p \sqsubseteq \top$, i.e., $\top$ is the greatest element w.r.t. $\sqsubseteq$. One can extend $\sqsubseteq$ pointwise to state transformers f, g by $f \sqsubseteq g \Leftrightarrow_{df} \forall \sigma.\, f(\sigma) \sqsubseteq g(\sigma)$.

Moreover separating conjunction $*$ on sets of states p, q is given by

$$p * q =_{df} \begin{cases} \{\sigma_1 \bullet \sigma_2 :\ \sigma_1 \# \sigma_2,\ \sigma_1 \in p,\ \sigma_2 \in q\} & \text{if } p, q \in \mathcal{P}(\Sigma) \\ \top & \text{otherwise}. \end{cases}$$

The semantics of these functions are given as forward strongest postcondition state transformers. A proper definition of $*$ on such functions, like in the relational case, leads to the problem that it generally does not entail associativity. The reason for this is that for obtaining strongest postconditions one would require distributivity of $*$ over arbitrary intersections which does only hold for precise assertions. The approach avoids this by directly defining a property called *locality* that is used to model the behaviour of the frame property and safety monotonicity. It is given for a state transformer f by

$$\sigma_1 \# \sigma_2 \ \Rightarrow\ f(\sigma_1 \bullet \sigma_2) \sqsubseteq f(\sigma_1) * \{\sigma_2\}. \tag{4.7}$$

All state transformers that satisfy locality are called *local actions*. This property has similar behaviour as our relational version of the frame property. It states that f locally acts on σ_1 while leaving σ_2 unchanged. A difference can be seen in the handling of program faults. In the case when $f(\sigma_1) = \top$, i.e., σ_1 is not safe for f, the right-hand side of the locality property will evaluate to $f(\sigma_1) * \{\sigma_2\} = \top * \{\sigma_2\} = \top$. In this case it is trivially satisfied, i.e., an explicit assumption of involved safe states is not needed. In the relational case it is asserted that σ_1 represents a safe state and hence any execution starting from it will not lead to program abortion. Both treatments handle the possibility of faulting in a demonic fashion, i.e., states that assert successful and aborting executions are excluded from the treatment of local actions and relations with the frame property.

As further work to this it would be interesting to investigate the characterisations of *footprints* given in [RG08] within the relational setting. Footprints are elements of an underlying separation algebra that are essential for a complete specification of the behaviour of local actions.

[1] $\top$ does not denote the universal relation in this context.

Another approach [HHM+11] involves a more meaningful structure given by so-called monotone predicate transformers, i.e., functions of type $\mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$. Differently from the approach of local actions, the semantics are dually provided as backward weakest precondition predicate transformers. This yields a definition of separating conjunction directly on such functions as in the case of relations:

$$\begin{aligned}(F_1 * F_2)(Y) &=_{df} \bigcup\{F_1(Y_1) * F_2(Y_2) : Y_1 * Y_2 \subseteq Y\}\,,\\ (F_1 \,;\, F_2)(Y) &=_{df} F_1(F_2(Y))\,,\\ Id\,(Y) &=_{df} Y.\end{aligned}$$

Note that $*$ is used in the first equation on sets of states on the right-hand side and in an overloaded form on functions on the left-hand side. A locality property for predicate transformer F is given for a set of states P by

$$F(P) = \bigcup\{F(X) * R : X * R = P\}\,. \tag{4.8}$$

The set X describes the set of safe states for the function F while R denotes again the untouched set of substates w.r.t. P. Note that predicate transformers entail more expressiveness than state transformers. Interestingly, it has been shown in [HHM+11] that Equation (4.8) is equivalent to the more compact form:

$$F * Id = F\,.$$

This means in particular that each execution of the program F can be replaced by one that only operates on the necessary and possible smaller part of the state while the rest of it remains unchanged (abstractly denoted by the function Id). It would be interesting to obtain a similar characterisation in the relational model. In the following, we derive some formulations and relationships to the semantics of the above formula by the use of the relational setting. Hence, we denote by the operator $*$ in what follows its relational version.

First remember that e as defined before Lemma 4.3.6 is the unit of $*$ and $e \subseteq I$. Thus we can immediately conclude:

Lemma 4.3.30 *For arbitrary relation C we have $C \subseteq C * I$.*

Proof. By isotony, we infer $C = C * e \subseteq C * I$. □

The other inclusion, i.e., $C * I \subseteq C$, does not generally hold. For this we need an additional assumption about C. This inequation can be derived from a stronger form of test preservation given in Definition 4.3.20:

$$\lhd\,;\,(C \times r)\,;\,\#\ \subseteq\ C\,;\lhd\,;\,(I \times r)\,. \tag{4.9}$$

The semantics of this inequation in a total correctness treatment is as follows: when C is executed on a part of the state such that the remainder of the state satisfies r one can also run C on the complete state and still obtains an r-part in the final state. A partial correctness interpretation would not exclude initial unsafe states, i.e., aborting executions. Note that one obtains from Equation (4.9) by isotony the specialisation of Definition 4.3.20 to $K = I$. Conceptually, Equation (4.9) assumes the existence of local executions of C leaving state portions of r unchanged. The relational frame property conversely talks about the structure of C in that it can be divided into subexecutions. Finally, we summarise:

Lemma 4.3.31 *$C * I \subseteq C$ iffs Equation (4.9) holds for all tests r.*

The proof can be found in Appendix A. It can be seen that there are some proof-theoretic relationships between our relational definitions for proving soundness of the frame rule and the concept of the locality condition $C * I \subseteq C$. However, for that particular domain it does not seem very realistic to relate locality to that form of test preservation. By using the compensator I, one implicitly assumes that the underlying separation algebras handle the involved side conditions of the frame rules, e.g., by permission equipped variables. In that case the preservation property should always be satisfied since it is used as an abstraction of the property that a command C does not modify the free variables of an assertion r.

Finally, we compare the structure of the relational proofs with the corresponding ones in [HHM+11]. First, note that predicate transformers are ordered with the reversed pointwise inclusion order, i.e., $F \sqsubseteq G \Leftrightarrow \forall X : F(X) \supseteq G(X)$. It was shown that the frame rule is equivalent to the inequation

$$(F * G) ; H \sqsubseteq (F ; H) * G$$

for adequate predicate transformers F, G, H using a generalised characterisation of Hoare triples. This inequation, also called the *small exchange law*, was previously introduced within the approach of *concurrenct Kleene algebras* [HMSW11]. We will elaborate on this algebraic structure later in Section 4.4.3.

In the relational setting, that inequation is not generally valid using the standard subset order on relations. However, as can be seen in Lemma 4.3.27 a structurally similar variant has been established with the pointfree version of the frame property. By choosing the compensator $K = I$ this yields for tests p, r and a relation C with the frame property:

$$\begin{aligned} p \subseteq \ulcorner C \quad &\Rightarrow \quad (p * r) ; C \subseteq (p ; C) * r \,, \\ p \subseteq \mathsf{safe}\,(C) \quad &\Rightarrow \quad (p * \widetilde{r}) ; C \subseteq (p ; C) * \widetilde{r} \,. \end{aligned}$$

Although the test p is restricted in the premise of the implications, the relational proofs of the frame rule become as simple as the one for the predicate transformer approach in [HHM+11]. As a further remark, the approach of [HHM+11] requires special functions for the semantics of Hoare triples. They are called *best predicate transformers* and are used as an adequate substitute for assertions. Intuitively, these functions simulate the allocation of resources that are characterised by pre- and postconditions. In the relational calculus, assertions can simply be handled with the subalgebra of tests. Another advantage of the relational approach is that $*$ distributes over arbitrary suprema while this is not the case for predicate transformers. By monotonicity one can only obtain a super-distributivity law for that treatment.

4.4 Applications to Concurrency

In Definition 4.3.5 we provided a general operation $*$ on arbitrary relations that, depending on the underlying separation algebra, captures as a special case the semantics of the separating conjunction. This extended operation allowed relational formulations that characterised the behaviour and structure of separation logical commands in a sequential setting. As already mentioned, one can also interpret general $*$-products $C * D$ of relations C, D as their parallel execution on $\#$-related parts of resources. This will be the main topic of this section. We start by presenting some central concepts and proof rules of concurrent separation logic (CSL) [Bro07, O'H07]. CSL has proved to be an effective methodology for scalable reasoning about concurrency. Moreover, we provide relational abstractions of this and pointfree validity proofs as in [DM12a, DM14]. As a next step we derive several relationships to other approaches that also involve algebraic semantics by modelling concurrent composition as separation of programs. This yields, besides further concrete applications for the presented relational abstractions, also new insights for future considerations about relations and concurrency reasoning.

4.4.1 Relations and Concurrent Separation Logic

We start this section with concrete definitions of the semantics of CSL and briefly explain the concepts of the concurrency extension of separation logic. After this relational denotations and formulations will be provided to model effects of separating resources within concurrent programs. These yield a fully pointfree proof of a central inference rule in CSL that we introduce in the sequel.

CSL is an extension of its sequential version with additional concepts and proof rules for reasoning about pointer manipulating programs in a concurrent environ-

ment [O'H07]. A soundness proof of this logic in a partial correctness treatment is provided in [Bro07] that uses a trace-based denotation with interleaving semantics. Starting from this approach we derive in the sequel relational abstractions of this. We begin by introducing extended separation logic triples in proof rules of CSL that come with an additional construct for controlling access to shared resources in a concurrent environment. It is called the *resource context* and is denoted by Γ. It is used to safeguard identifiers or program variables that belong to a critical resource r with a corresponding invariant. Such resources can usually be accessed concurrently and therefore require some care to avoid non-deterministic behaviour. Contexts are appended to formulas involving separation logic triples, i.e.,

$$\Gamma \vdash \{p\}\, C\, \{q\}\,.$$

Programs C that use resources of Γ need to maintain the invariants involved after execution. As a concrete example of such a resource one can think of a shared queue, implemented as a list. The invariant can be given by $\mathsf{list}\ \alpha\ \mathsf{i}$ (cf. Example 2.1.1) and the protected variable by i. The ghost variable α is not protected by Γ since it is not visible within the program itself. Now any execution accessing the list by a dequeue or enqueue of elements needs to maintain the linking structure of the list.

Generally, soundness of the proof rules in CSL require that resource invariants are precise assertions. Another soundness proof that comes with a weaker requirement can be found in [Vaf11]. We will not go into any further details on resource contexts since intuition suffices to grasp the central ingredient of CSL, namely the *concurrency rule*, which allows a parallel composition of programs:

$$\frac{\Gamma \vdash \{p_1\}\, C\, \{q_1\} \quad \Gamma \vdash \{p_2\}\, D\, \{q_2\}}{\Gamma \vdash \{p_1 * p_2\}\, C \parallel D\, \{q_1 * q_2\}} \qquad \text{(concrule)}$$

where p_i, q_i are separation logic assertions and C, D denote commands. There are additional side conditions for this proof rule on the variables involved: $\mathrm{FV}(p_1, q_1) \cap \mathrm{MV}(D) = \mathrm{FV}(p_2, q_2) \cap \mathrm{MV}(C) = \emptyset$, and that $\mathrm{FV}(C) \cap \mathrm{MV}(D)$ and $\mathrm{FV}(D) \cap \mathrm{MV}(C)$ need to be subsets of the identifiers protected by the resource context Γ. The former side condition is similar as in the frame rule and requires that assertion variables of the untouched resources should not be modified. The latter condition asserts that modified variables occurring free in a parallel program must be used inside a critical region, i.e., protected by the resource context. Otherwise races can occur, i.e., simultaneous accesses of parallel running programs to the same resources. Such races may induce uncontrolled behaviour and therefore need to be excluded. The concurrency rule informally allows, under the described circumstances, to compose programs in parallel, each of them running on $*$-separated regions of storage.

The semantics of CSL provided in [Bro07] is quite complex. In what follows we briefly present the basics that are needed to state the central theorem of that approach for showing soundness of the concurrency rule. For the trace semantics of CSL, one defines that a trace, denoted by α, is a non-empty finite or infinite sequence of *actions*, that are denoted by λ. Such traces are also called *action traces*. Concatenation of traces α_1 and α_2 is written $\alpha_1\alpha_2$. Moreover, a special action *abort* is introduced that is a left annihilator, i.e., $\alpha_1\, abort\, \alpha_2 = \alpha_1\, abort$. Intuitively, it signalises the occurrence of a race. Actions λ are given by a subset of the standard separation logic commands (cf. Section 2.2), i.e., heap lookup, mutation, allocation, disposal and store variable assignment.

Additionally, for the treatment of critical resources, special *resource actions* are defined by $try(r)$, $acq(r)$ and $rel(r)$, i.e., waiting for a resource r to be available, acquiring and releasing it, respectively. By the use of actions λ, CSL defines a *resource enabling* relation $\xrightarrow{\lambda}$ on pairs (A_1, A_2) of disjoint resource name sets A_1, A_2. Its general purpose is to track in the first component of the pairs the critical resources that are acquired or released by any executed action while A_2 includes resource names that are acquired e.g., by other concurrently running threads. The enabling relation is defined by

$$
\begin{array}{ll}
(A_1, A_2) \xrightarrow{acq(r)} (A_1 \cup \{r\}, A_2) & \text{if } r \notin A_1 \cup A_2\,, \\
(A_1, A_2) \xrightarrow{rel(r)} (A_1 - \{r\}, A_2) & \text{if } r \in A_1\,, \\
(A_1, A_2) \xrightarrow{\lambda} (A_1, A_2) & \text{if } \lambda = try(r) \text{ or } \lambda \text{ is not a resource action.}
\end{array}
$$

Informally, $acq(r)$ adds a resource name r to A_1 if it is not contained in any of the sets A_i while $rel(r)$ deletes r from A_1 only if it is contained in A_1. All other actions λ will leave such pairs unchanged. The resource enabling relation can be generalised to arbitrary traces α. This allows the definition of an operator to interleave action traces. Assume resource name sets A_i for $i \in \{1, 2\}$ that denote initial acquired sets of resources that are held by programs executing the trace α_i. For traces α_1, α_2 we write $\alpha_1 \;{}_{A_1}\|_{A_2}\; \alpha_2$ for the set of all possible interleavings of α_1 and α_2 w.r.t. the resources name sets A_i. Before we give a concrete definition of the interleaving operation we require a notion for the case when actions might interfere with each other's execution: An action λ_1 *interferes with* another action λ_2 iff

$$\mathrm{FV}(\lambda_1) \cap MV(\lambda_2) \neq \emptyset \;\;\vee\;\; \mathrm{FV}(\lambda_2) \cap MV(\lambda_1) \neq \emptyset\,,$$

i.e., one or both actions can produce a race which may yield non-deterministic behaviour by modifying the free variables of the parallel executed action. The definition of the sets $\mathrm{FV}(\lambda)$ and $MV(\lambda)$ for an action λ can be found in Appendix A.3 (by identifying λ with commands over which the actions range).

Finally the set of interleaved traces of α_1, α_2 can be recursively given as in Figure 4.3 where ε denotes the empty sequence. The first two equations are the base cases where one of the parallel executed program can not execute any further action. We have e.g., $\alpha_1 \;{}_{A_1}\|_{A_2}\; \varepsilon = \{\alpha_1\}$ only if $(A_1, A_2) \xrightarrow{\alpha_1} (A_1', A_2)$, i.e., α_1 can be executed with the initial set A_1 and ends with A_1'. Otherwise $\alpha_1 \;{}_{A_1}\|_{A_2}\; \varepsilon = \emptyset$. The last case recursively interleaves the leading actions λ_1, λ_2 with the subsequent traces α_1 and α_2 by the use of the resource enabling relation. The sets A_i' again denote the acquired resources after the execution of each λ_i.

$$
\begin{aligned}
\alpha_1 \;{}_{A_1}\|_{A_2}\; \varepsilon &=_{df} \{\alpha_1 : (A_1, A_2) \xrightarrow{\alpha_1} (A_1', A_2)\}, \\
\varepsilon \;{}_{A_1}\|_{A_2}\; \alpha_1 &=_{df} \{\alpha_2 : (A_2, A_1) \xrightarrow{\alpha_2} (A_2', A_1)\}, \\
\lambda_1\alpha_1 \;{}_{A_1}\|_{A_2}\; \lambda_2\alpha_2 &=_{df} \{\, \mathit{abort} : \lambda_1 \text{ interferes with } \lambda_2\} \\
&\quad \cup \{\lambda_1\beta : (A_1, A_2) \xrightarrow{\lambda_1} (A_1', A_2),\ \beta \in (\alpha_1 \;{}_{A_1'}\|_{A_2}\; \lambda_2\alpha_2)\} \\
&\quad \cup \{\lambda_2\beta : (A_2, A_1) \xrightarrow{\lambda_2} (A_2', A_1),\ \beta \in (\lambda_1\alpha_1 \;{}_{A_1}\|_{A_2'}\; \alpha_2)\}.
\end{aligned}
$$

Figure 4.3: Recursive definition of interleaving traces.

Note that the trace *abort* above signalises the possibility of a race and can also occur within the merge in the recursive cases. The subsequent sets recursively build all other possible traces respecting the resource name sets A_1, A_2. Programs in CSL are defined to initially start with the empty set of resource names, i.e., at the beginning there are no resources acquired and hence $A_1 = A_2 = \emptyset$. We abbreviate ${}_{\emptyset}\|_{\emptyset}$ to $\|$. Moreover, we set $MV(\alpha) =_{df} \bigcup_{1 \leq i \leq n} \mathrm{MV}(\lambda_i)$ for a trace $\alpha = \lambda_1 \ldots \lambda_n$, i.e., the set of variables that are modified within any action of α. Using this, we denote by $s \backslash MV(\alpha)$ the substore of a store s whose domain equals $dom(s) - MV(\alpha_2)$.

Now, the key ingredient for establishing soundness of the concurrency rule is a *local enabling* relation given by $\sigma \xrightarrow[\Gamma]{\alpha} \sigma'$. It states that the local execution of a trace α from an initial state σ is enabled and will end in the final state σ'. In particular, it needs to be consistent with the resource context Γ in the sense that only accesses to unprotected identifiers (program variables) or those of acquired resources can occur while respecting the corresponding resource invariants. Concretely, the states σ involved are of the form (s, h, A), where A denotes the set of resource names that a program holds at this state. As particular cases, one defines for arbitrary state σ, e.g.,

$$
\sigma \xrightarrow[\Gamma]{\mathsf{skip}} \sigma, \quad \sigma \xrightarrow[\Gamma]{\mathit{abort}} \mathsf{abort}, \quad (s, h, A) \xrightarrow[\Gamma]{[l]:=v} (s, (l, v) \,|\, h, A) \quad \text{if } l \in dom(h),
$$

where $l \in \mathit{Addresses}$ and $v \in \mathit{Values}$. For a complete definition of this relation involving the remaining concrete actions we refer to [Bro07]. Note that the validity statement $\models$ (cf. Section 2.1) of separation logic assertions can easily be defined on

the resource name set of extended states. For abbreviation, we also write (s, h) for states with an empty set of acquired resources, i.e., $(s, h, \emptyset)$.

Finally, we are able to state the *parallel decomposition theorem* of [Bro07] from which we start the derivation of relational abstractions. By $\mathsf{tr}(C)$ we denote the set of traces of a syntactically given command C. Again we refer to [Bro07] for the concrete definitions to derive the corresponding traces of such commands.

Theorem 4.4.1 (Parallel Decomposition) *Assume for syntactic commands C, D that the sets $\mathrm{FV}(C) \cap \mathrm{MV}(D)$ and $\mathrm{FV}(D) \cap \mathrm{MV}(C)$ are subsets of the identifiers protected by a resource context Γ. Moreover, let $\alpha \in (\alpha_1 \parallel \alpha_2)$ where $\alpha_1 \in \mathsf{tr}(C)$, $\alpha_2 \in \mathsf{tr}(D)$ and $h = h_1 \cup h_2$ with $dom(h_1) \cap dom(h_2) = \emptyset$.*

- *If $(s, h) \xrightarrow[\Gamma]{\alpha} \mathsf{abort}$ then $(s \backslash MV(\alpha_2), h_1) \xrightarrow[\Gamma]{\alpha_1} \mathsf{abort}$ or $(s \backslash MV(\alpha_1), h_2) \xrightarrow[\Gamma]{\alpha_2} \mathsf{abort}$,*
- *if $(s, h) \xrightarrow[\Gamma]{\alpha} (s', h')$ then $(s \backslash MV(\alpha_2), h_1) \xrightarrow[\Gamma]{\alpha_1} \mathsf{abort}$ or $(s \backslash MV(\alpha_1), h_2) \xrightarrow[\Gamma]{\alpha_2} \mathsf{abort}$ or there are heaps h_1', h_2' with $h' = h_1' \cup h_2'$, $dom(h_1') \cap dom(h_2') = \emptyset$ and*
  - *$(s \backslash MV(\alpha_2), h_1) \xrightarrow[\Gamma]{\alpha_1} (s' \backslash MV(\alpha_2), h_1')$,*
  - *$(s \backslash MV(\alpha_1), h_2) \xrightarrow[\Gamma]{\alpha_2} (s' \backslash MV(\alpha_1), h_2')$.*

The assumption on the free and modified variables of the involved commands are required to guarantee that critical variables are protected by the resource context. More interestingly, the central conclusion of this theorem is that any interleaving of $\alpha_1 \parallel \alpha_2$ basically only depends on the locally executed traces α_1, α_2 besides the cases of abortion. This gives us enough information to develop relationships between the abstract formalised $*$-operation on relations modelling denotations for syntactic commands C, D and their interleaved parallel execution $C \parallel D$.

First, assume a resource context Γ. We define relational abstractions of syntactic CSL commands parameterised by Γ as follows:

$$[\![C]\!]_\Gamma =_{df} \{(\sigma, \sigma') : \sigma \xrightarrow[\Gamma]{\alpha} \sigma', \alpha \in \mathsf{tr}(C), \alpha \text{ is finite}\}.$$

The parametrisation w.r.t. Γ is required due to the structure of the proof rules. They globally assume the same resource context for all involved triples. We only consider finite traces for $[\![C]\!]_\Gamma$ since we stay with the developed approach of identifying non-termination with divergence. Thus, $(\sigma, \sigma') \in [\![C]\!]_\Gamma$ iff there exists some finite trace $\alpha \in \mathsf{tr}(C)$ that enables the transition. The set of traces of the parallel execution $C \parallel D$ is given by $\mathsf{tr}(C \parallel D) =_{df} \bigcup\{\alpha_1 \parallel \alpha_2 : \alpha_1 \in \mathsf{tr}(C), \alpha_2 \in \mathsf{tr}(D)\}$. Hence, we

immediately infer

$$[\![C \parallel D]\!]_\Gamma = \{(\sigma, \sigma') : \sigma \xrightarrow[\Gamma]{\alpha} \sigma',\ \alpha \in \alpha_1 \parallel \alpha_2,\ \alpha_1 \in \mathsf{tr}(C),\ \alpha_2 \in \mathsf{tr}(D),\ \alpha \text{ is finite}\}\,.$$

Finally, we can derive from Theorem 4.4.1 pointfree relational abstractions. First, by the use of the concepts provided in Section 4.3 we can model changes on the variables involved by adequate compensator relations. Concretely, we can define e.g., for a syntactical command C the compensator H_C by

$$\begin{aligned}(s,h,A)\ H_C\ (s',h',A') \Leftrightarrow_{df} \mathrm{MV}(C) \subseteq dom(s) \wedge \mathrm{MV}(C) \subseteq dom(s') \\ \wedge\ h = h' \wedge A = A'\,.\end{aligned} \tag{4.10}$$

It is not difficult to see that H_C is in fact a compensator. It changes the modified variables of a syntactic command C arbitrarily while maintaining the heap and resource name components. The modelling of a substore $s\backslash\mathrm{MV}(C)$ w.r.t. a command C with stores where $\mathrm{MV}(C)$ is arbitrarily changed will not invalidate the approach since both treatments will generate program faults in the case when the variable conditions are not satisfied. Clearly, this concept can be abstracted as before. The condition on the variables protected by the environment cannot be modelled. We suppose that this could be incorporated by an adequate modification of the concrete model of [Bro07] by variables equipped with permissions. The details remain as future work. For simplicity we suppose for the purpose of abstraction that this condition is implicitly satisfied in the sequel as it is required for the concurrency rule (concrule). Moreover, program states of the form (s,h,A), extended by resource name sets A, can also be treated within the setting of multi-unit separation algebras. The concrete combinability relation is defined by $(s,h,A) \# (s',h',A') \Leftrightarrow_{df} s = s' \wedge dom(h) \cap dom(h') = \emptyset \wedge A \cap A' = \emptyset$. As in the case of heaps, resource name sets are treated by disjoint union and hence with each store s a unit is given by $(s,\emptyset,\emptyset)$.

For better readability, we will omit in the following definitions and calculations the braces $[\![\_]\!]_\Gamma$ and denote by C, D and $C \parallel D$ the corresponding relational denotations in $\Sigma_\perp \times \Sigma_\perp$ due to a partial correctness treatment instead of the syntactical commands.

Definition 4.4.2 (Relational Decomposition)
Assume commands C, D and associated compensators K_C, K_D, respectively. Then pointfree formulations of Theorem 4.4.1 are obtained by

$$C \parallel D \ \subseteq\ ((K_D\,;C) * (K_C\,;D))\,;\perp\ \cup\ (\mathsf{safe}\,(C) * \mathsf{safe}\,(D))\,;(C \parallel D)$$

and

$$(\mathsf{safe}\,(C) \times \mathsf{safe}\,(D))\,;\rhd\,;(C \parallel D)\ \subseteq\ (K_D\,;C\,;K_D\ \times\ K_C\,;D\,;K_C)\,;\rhd\,.$$

The first inequation describes the fact that either the command C or D or both will abort, e.g., due to a race or memory fault. Note that by Lemma 4.3.7 the relation $((K_D\,;C)*(K_C\,;D))\,;\bot$ includes all these cases, since it is equal to $(K_D\,;C\,;\bot)*(K_C\,;D)\cup(K_D\,;C)*(K_C\,;D\,;\bot)$. Assuming states σ,σ' we provide for a better intuition the pointwise form for one of the relations:

$$\begin{aligned}&\sigma\ (K_D\,;C\,;\bot)*(K_C\,;D)\ \sigma'\\ \Leftrightarrow\ &\exists\,\sigma_1,\sigma_2,\sigma_2',\tau_1,\tau_2:\ \sigma_1\#\sigma_2\ \wedge\ \sigma=\sigma_1\bullet\sigma_2\ \wedge\ \sigma_1\ K_D\ \tau_1\ \wedge\ \sigma_2\ K_C\ \tau_2\\ &\wedge\ \exists\,\alpha_1\in\mathsf{tr}(C):\ \tau_1\xrightarrow[\Gamma]{\alpha_1}\sigma_\bot\ \wedge\ \exists\,\alpha_2\in\mathsf{tr}(D):\ \tau_2\xrightarrow[\Gamma]{\alpha_2}\sigma_2'\ \wedge\ \sigma'=\sigma_\bot\,.\end{aligned}$$

In the non-aborting case, the initial state can be split into two substates that are safe for C and D, i.e., the execution will not lead to a program abortion, and any interleaved execution of them can be split into two subexecutions acting on the substates. This case is stated in a style similar to the case of the frame property in Definition 4.3.16 using products of relations. Note that in CSL syntactical commands C satisfy $[\![\,C\parallel\mathsf{skip}\,]\!]_\Gamma=[\![\,C\,]\!]_\Gamma$ while a relational abstraction of skip would translate to the identity relation I. By setting $D=I$ and $K_D=I$, we get the partial correctness frame property as a special case of the second inequation of Definition 4.4.2 since $\mathsf{safe}\,(I)=\neg\bot$ and compensators are transitive. Hence, Definition 4.3.16 can be seen as an extension of the frame property. Moreover, we can immediately infer from this the following result.

Lemma 4.4.3 *Assume commands C,D and associated compensators K_C,K_D, respectively. Suppose C,D satisfy Definition 4.4.2. Then for all tests p_1,p_2 we have*

$$\begin{aligned}p_1\ \subseteq\ \mathsf{safe}\,(C)\ \wedge\ p_2\ \subseteq\ \mathsf{safe}\,(D)\ \ \Rightarrow\\ (p_1*p_2)\,;(C\parallel D)\ \subseteq\ (p_1\,;K_D\,;C\,;K_D)*(p_2\,;K_C\,;D\,;K_C)\,.\end{aligned}$$

A proof of this can be given similarly as for Lemma 4.3.27. We continue with a relational formulation of the variable side conditions of the concurrency rule. Unfortunately the general notion of preservation (cf. Section 4.3.2) cannot be directly applied in its original form for this setting. The reason for this is that Equation (4.4) only considers pairs of relations with combinable initial and final states. This cannot be established in $*$-products of arbitrary $;$-composed relations. It is therefore put as an assumption e.g., in Lemma 4.3.23. We will use a stronger variant that implies Equation (4.4) and only involves a compensator and hence not the corresponding relational denotation of the command.

Definition 4.4.4 (Strong preservation)
A compensator K_C of a command C *strongly preserves* a test p iff $p\,;K_C\ \subseteq\ K_C\,;p$.

This corresponds to a standard and well-known relational characterisation of invariants in the literature. It states that p has to hold before and after the execution of K_C. For concrete separation logic assertions, assume $K_C = H_C$ as in (4.10). Then by preservation, C will modify at most irrelevant variables of an assertion p. By Lemma 4.3.26 this form of preservation is equivalent to

$$p\,;K_C \subseteq \top\,;p\,, \tag{strong pres}$$

where $\top$ denotes the universal relation. As before, this form of preservation is also closed under union and relation composition. Moreover, any test q strongly preserves any test p. In particular, I strongly preserves any test p.

For a pointfree and calculational proof of concurrency rule, it remains as a last step to provide a characterisation of the triples $\Gamma \vdash \{p\}\,C\,\{q\}$ in CSL. Therefore we consider the concrete semantics according to [Bro07].

Definition 4.4.5 (CSL Triples)
Assume a resource context Γ, assertions p, q and a syntactical command C. Then $\Gamma \vdash \{p\}\,C\,\{q\}$ is valid if for all traces $\alpha \in \mathsf{tr}(C)$ and all states $(s,h), \sigma'$ we have that $(s,h) \models p$ and $(s,h) \xrightarrow[\Gamma]{\alpha} \sigma'$ implies $\neg((s,h) \xrightarrow[\Gamma]{\alpha} \mathsf{abort})$ and $\sigma' \models q$. In particular, all free variables of the syntactical command C and resource invariants of Γ without the identifiers protected by Γ need to be contained in $dom(s)$.

We implicitly assume in the sequel that the additional constraint in Definition 4.4.5 on the variables involved are satisfied. By this, we can reuse Definition 4.2.6, due to abstraction, also for the triples of CSL by the use of the relational abstractions $[\![\_]\!]_\Gamma$ of commands and $[\![\_]\!]$ for assertions (cf. Section 4.2) to obtain appropriate tests. By Lemma 4.3.26 we therefore concretely define

$$\Gamma \vdash \{p\}\,C\,\{q\} \;\Leftrightarrow_{df}\; \widetilde{[\![p]\!]}\,;[\![C]\!]_\Gamma \subseteq \top\,;\widetilde{[\![q]\!]}\,. \tag{CSL triples}$$

Now, we can give an abstract and pointfree proof of the concurrency rule of CSL.

Theorem 4.4.6 (Concurrency Rule) *Assume a resource context Γ and syntactic commands C, D with relational denotations $[\![C]\!]_\Gamma, [\![D]\!]_\Gamma$ satisfying Definition 4.4.2 with corresponding compensators K_C, K_D. Moreover, assume that K_C strongly preserves tests p_2, q_2 and K_D strongly preserves tests p_1, q_1. Then the concurrency rule is valid, i.e.,*

$$\frac{\Gamma \vdash \{p_1\}\,C\,\{q_1\} \quad \Gamma \vdash \{p_2\}\,D\,\{q_2\}}{\Gamma \vdash \{p_1 * p_2\}\,C \parallel D\,\{q_1 * q_2\}}\,.$$

Proof. For easier readability we omit the brackets $[\![\_]\!]_\Gamma, [\![\_]\!]$. First, Lemma 4.2.8, the assumptions and (CSL triples) imply $\widetilde{p_1} \subseteq \mathsf{safe}(C)$ and $\widetilde{p_2} \subseteq \mathsf{safe}(D)$. By

Lemma 4.3.8, Lemma 4.4.3, K_C strongly preserves p_2 and K_D strongly preserves p_1 w.r.t. (strong pres), assumptions and (CSL triples), isotony, K_C strongly preserves q_2 and K_D strongly preserves q_1 w.r.t. (strong pres), isotony, Lemma 4.3.25(b), and Lemma 4.3.8:

$$\begin{array}{ll} & \widetilde{(p_1 * p_2)} \,;\, (C \parallel D) \\ = & (\widetilde{p_1} * \widetilde{p_2}) \,;\, (C \parallel D) \\ \subseteq & (\widetilde{p_1} \,;\, K_D \,;\, C \,;\, K_D) * (\widetilde{p_2} \,;\, K_C \,;\, D \,;\, K_C) \\ \subseteq & (\top \,;\, \widetilde{p_1} \,;\, C \,;\, K_D) * (\top \,;\, \widetilde{p_2} \,;\, D \,;\, K_C) \\ \subseteq & (\top \,;\, \widetilde{q_1} \,;\, K_D) * (\top \,;\, \widetilde{q_2} \,;\, K_C) \\ \subseteq & (\top \,;\, \widetilde{q_1}) * (\top \,;\, \widetilde{q_2}) \\ \subseteq & \top \,;\, (\widetilde{q_1} * \widetilde{q_2}) \\ \subseteq & \top \,;\, \widetilde{(q_1 * q_2)} \,. \end{array}$$

□

As in the case of the frame rule it can be seen that an algebraic proof of validity of the concurrency rule is not difficult. Moreover, since the proof is expressible in first-order logic this further allows an automated and mechanised soundness proof of this inference rule. A further advantage of our approach is that it is not difficult to obtain from the provided definitions further formulations for a total correctness treatment of CSL.

Note that in the case of separation algebras with an additional permission structure on the store variable as in [BCY06] we can use the compensator I and simplify the concurrency rule and its pointfree assumptions used in the proof.

Corollary 4.4.7 *Assume relations C, D has the compensator I. Suppose C, D satisfy Definition 4.4.2. Then for all tests p_1, p_2 we have*

$$p_1 \subseteq \mathsf{safe}\,(C) \wedge p_2 \subseteq \mathsf{safe}\,(D) \;\Rightarrow\; (p_1 * p_2) \,;\, (C \parallel D) \subseteq (p_1 \,;\, C) * (p_2 \,;\, D)\,.$$

Corollary 4.4.8 *Assume a resource context Γ and syntactic commands C, D with relational denotations $[\![C]\!]_\Gamma, [\![D]\!]_\Gamma$ and compensator $K_C = K_D = I$ satisfying Definition 4.4.2. Then the concurrency rule is valid, i.e.,*

$$\frac{\Gamma \vdash \{p_1\}\, C\, \{q_1\} \quad \Gamma \vdash \{p_2\}\, D\, \{q_2\}}{\Gamma \vdash \{p_1 * p_2\}\, C \parallel D\, \{q_1 * q_2\}}\,.$$

This is the general form of the concurrency rule used in the literature (e.g. [COY07, HMSW11, HHM+11]). A semantic and abstract approach for the concurrency rule can also be found in [COY07]. That treatment is based on special state transformer w.r.t. elements of an abstract separation algebra. This makes that approach more general

and not specific to the state model of CSL. Moreover, they handle races differently than the approach of [Bro07] which is the base of our treatment. However, there are relationships to our approach, e.g., the simplified version of parallel decomposition in [COY07] closely corresponds to our pointfree substitute in Corollary 4.4.7. In both approaches any interleaved parallel execution depends on the local executions of the individual commands themselves.

4.4.2 Disjoint Concurrency

In the previous section we used for a proof of the concurrency rule relational denotations of the form $[\![C \parallel D]\!]_\Gamma$ to model interleaved concurrency. Related approaches that provide algebraic abstractions to concurrency (e.g. [HMSW11, HHM+11]) define a special operation for modelling concurrent composition. The central question that arises for our relational treatment is: *Can we use the generalised $*$ - operation to model concurrent composition in an interleaved fashion ?*

For simplicity we assume in the sequel separation algebras that incorporate variable preservation by permission structures and thus use I as a compensator for any command. By this, Definition 4.4.2 yields

$$[\![C \parallel D]\!]_\Gamma \subseteq ([\![C]\!]_\Gamma * [\![D]\!]_\Gamma) ; \bot \cup (\mathsf{safe}\,([\![C]\!]_\Gamma) * \mathsf{safe}\,([\![D]\!]_\Gamma)) ; [\![C \parallel D]\!]_\Gamma$$

and

$$(\mathsf{safe}\,([\![C]\!]_\Gamma) \times \mathsf{safe}\,([\![D]\!]_\Gamma)) ; \rhd ; [\![C \parallel D]\!]_\Gamma \subseteq ([\![C]\!]_\Gamma \times [\![D]\!]_\Gamma) ; \rhd .$$

By the definition of $*$ and isotony, this immediately implies that

$$[\![C \parallel D]\!]_\Gamma \subseteq [\![C]\!]_\Gamma * [\![D]\!]_\Gamma .$$

Unfortunately, the other inclusion does not follow generally. To see this, assume a resource context Γ and consider the concrete denotation of the right-hand side that is given by

$$\begin{aligned} [\![C]\!]_\Gamma * [\![D]\!]_\Gamma = \{ (\sigma_1 \bullet \sigma_2, \sigma_1' \bullet \sigma_2') : \exists\, \alpha : \sigma_1 \bullet \sigma_2 \xrightarrow[\Gamma]{\alpha} \sigma_1' \bullet \sigma_2' , \sigma_1 \# \sigma_2 , \sigma_1' \# \sigma_2' , \\ \sigma_1 \xrightarrow[\Gamma]{\alpha_1} \sigma_1', \alpha_1 \in \mathsf{tr}(C), \alpha_1 \text{ is finite}, \\ \sigma_2 \xrightarrow[\Gamma]{\alpha_2} \sigma_2', \alpha_2 \in \mathsf{tr}(D), \alpha_2 \text{ is finite} \} . \end{aligned}$$

The definition of $*$ does not include any details about the transition α from $\sigma_1 \bullet \sigma_2$ to $\sigma_1' \bullet \sigma_2'$. It can only be inferred that an independent execution of α_1 and α_2 on combinable or disjoint portions of the heap will lead to a composed final state. However, the denotation $[\![C \parallel D]\!]_\Gamma$ requires $\alpha \in \alpha_1 \parallel \alpha_2$ that can include traces

that have a race and therefore yields different executions. This means concretely that $*$-compositions of relations basically model disjoint concurrency with no interference or at most successfully synchronised interaction on shared resources between parallel programs. Executions that might include races are excluded and treated as non-termination erroneous programs.

There are still concrete applications for this kind of concurrency (e.g. [O'H07]). In the sequent we continue by abstract consideration and consequences for the case of disjoint concurrency. We formulate them for simplicity in terms of arbitrary relations $C, D \subseteq \Sigma \times \Sigma$ in a total correctness fashion as in [DM12a, DM14]. The resulting properties can be further used to prove a variant of concurrency rule. First, we start by a version of the decomposition theorem w.r.t. the generalised $*$-operator.

Definition 4.4.9
We define that relations C, D have the *disjoint decomposition property* iff

$$(\ulcorner C \times \ulcorner D) \,;\, \rhd \,;\, (C * D) \;\subseteq\; (C \times D) \,;\, \rhd \;.$$

The semantics are as follows: whenever two combinable initial states σ_1 and σ_2 provide enough resources for the execution of commands C, D then each of them will be able to acquire its needed resource from the joined state $\sigma_1 \bullet \sigma_2$ without any interference. For concrete commands, consider the simple separation algebra based on heaps of Example 3.3.2. We define relational denotations for heap mutation commands and single cell heaps by

$$\begin{array}{l} [\![\, [l] := v \,]\!] \;=_{df}\; \{(h, (l, v) \,|\, h) : l \in \mathit{dom}(h)\}\,, \\ [\![\, l \mapsto - \,]\!] \;=_{df}\; \bigcup_{v \in \mathbb{N}} \{(\{(l, v)\}, \{(l, v)\})\}\,. \end{array} \tag{4.11}$$

Intuitively, the command $[l] := v$ ensures that after any terminating execution the heap cell at address l contains the value v while $l \mapsto -$ describes heaps containing a single cell at address l with arbitrary contents. In particular, we have $\ulcorner [\![\, [l] := v \,]\!] = [\![\, l \mapsto - \,]\!] * I$. By setting $C = [\![\, [1] := 2 \,]\!]$ and $D = [\![\, [2] := 1 \,]\!]$, one obtains an instance of Definition 4.4.9. As before we can immediately infer the following results.

Lemma 4.4.10 *Let relations C and D have the disjoint decomposition property and assume $p_1 \subseteq \ulcorner C \wedge p_2 \subseteq \ulcorner D$. Then $(p_1 * p_2) \,;\, (C * D) \subseteq (p_1 \,;\, C) * (p_2 \,;\, D)$.*

Corollary 4.4.11 *Let C and D have the disjoint decomposition property. Then*

$$\frac{[\, p_1 \,]\, C\, [\, q_1 \,] \quad [\, p_2 \,]\, D\, [\, q_2 \,]}{[\, p_1 * p_2 \,]\, C * D\, [\, q_1 * q_2 \,]}\;.$$

For a better intuition we further provide a counterexample involving commands that do not satisfy Definition 4.4.9.

Example 4.4.12 Consider

$$C =_{df} [\![[1] := 1]\!] \cup [\![[2] := 1]\!] \quad \text{and} \quad D =_{df} [\![[2] := 1]\!] \cup [\![[2] := 2]\!],$$

where $\cup$ can be interpreted by non-deterministic choice. Clearly, the commands show interference with each other, since both may access the same heap locations. In fact, C and D do not satisfy the concurrency property. Note that $\ulcorner C = \ulcorner D = \{(h,h) : 1 \in dom(h) \vee 2 \in dom(h)\}$. Now, choose heaps $h_1 = \{(1,0)\}$ and $h_2 = \{(2,0)\}$. By $(h_1,h_1) \in \ulcorner C$, $(h_2,h_2) \in \ulcorner D$ and $h = h_1 \bullet h_2$, we have $(h,h) \in \ulcorner(C * D)$. A possible execution of $C * D$ is e.g., $(h, \{(1,2),(2,1)\})$. Hence, $((h_1,h_2), \{(1,2),(2,1)\})$ is included in the left-hand side of the instantiated disjoint decomposition property but not in the right-hand side, since that relation only allows $((h_1,h_2), \{(1,1),(2,2)\})$. □

This example shows that non-determinism in combination with commands working on different heap locations may introduce undesired behaviour. This yields a characterisation of commands that rule out that behaviour. We will see in the sequent that a sufficient concept to guarantee this is provided by *preciseness* (cf. Section 3.2.3). For better readability, we abbreviate for a test p the formula $(\sigma,\tau) \in p$ by $\sigma \in p$ since one generally has $(\sigma,\tau) \in p \Leftrightarrow (\sigma,\sigma) \in p \wedge \sigma = \tau$. We repeat in the context of a separation algebra: A test p is called *precise* iff for all states σ, there exists at most one substate σ' for which $\sigma' \in p$, i.e.,

$$\forall \sigma, \sigma_1, \sigma_2 : (\sigma_1 \in p \wedge \sigma_2 \in p \wedge \sigma_1 \preceq \sigma \wedge \sigma_2 \preceq \sigma) \Rightarrow \sigma_1 = \sigma_2 . \tag{4.12}$$

It turned out that precise tests can also be defined by the use of the split and join relations with tests. We start by an intermediate structural result that facilitates the proof of Lemma 4.4.14.

Lemma 4.4.13 $\sigma_1 \in p \wedge \sigma_1 \preceq \sigma \Leftrightarrow \exists \sigma_2 : \ ((\sigma_1,\sigma_2),\sigma) \in (p \times I) ; \rhd$.

Proof. By definition of $\preceq$, logic, $\sigma_2 \in I \Leftrightarrow \mathsf{true}$, and definition of $\rhd$:

$$\begin{aligned}
& \sigma_1 \in p \wedge \sigma_1 \preceq \sigma \\
\Leftrightarrow\ & \sigma_1 \in p \wedge \exists \sigma_2 : \ \sigma_1 \# \sigma_2 \wedge \sigma_1 \bullet \sigma_2 = \sigma \\
\Leftrightarrow\ & \exists \sigma_2 : \ \sigma_1 \in p \wedge \sigma_2 \in I \wedge \sigma_1 \# \sigma_2 \wedge \sigma_1 \bullet \sigma_2 = \sigma \\
\Leftrightarrow\ & \exists \sigma_2 : \ ((\sigma_1,\sigma_2),\sigma) \in (p \times I) ; \rhd .
\end{aligned}$$

□

Lemma 4.4.14 *If a test p satisfies*

$$(p \times I)\,;\rhd\,;\lhd\,;(p \times I) \;\subseteq\; p \times I \tag{4.13}$$

then it is precise.

Proof. Using Lemma 4.4.13 we rewrite (4.12). Now by a logic step, Lemma 4.4.13 and $\lhd$ is the converse of $\rhd$, definition of ; and of tests and $\times$, and again a logic step

$$\begin{array}{ll}
& \forall\,\sigma,\sigma_1,\sigma_2 : (\sigma_1 \in p \;\wedge\; \sigma_2 \in p \;\wedge\; \sigma_1 \preceq \sigma \;\wedge\; \sigma_2 \preceq \sigma) \;\Rightarrow\; \sigma_1 = \sigma_2 \\
\Leftrightarrow & \forall\,\sigma_1,\sigma_2 : (\exists\,\sigma :\; \sigma_1 \in p \;\wedge\; \sigma_2 \in p \;\wedge\; \sigma_1 \preceq \sigma \;\wedge\; \sigma_2 \preceq \sigma) \;\Rightarrow\; \sigma_1 = \sigma_2 \\
\Leftrightarrow & \forall\,\sigma_1,\sigma_2 : (\exists\,\sigma :\; (\exists\,\tau_1 :\; ((\sigma_1,\tau_1),\sigma) \in (p \times I)\,;\rhd) \wedge \\
& \qquad\qquad (\exists\,\tau_2 :\; (\sigma,(\sigma_2,\tau_2)) \in \lhd\,;(p \times I)\,) \;\Rightarrow\; \sigma_1 = \sigma_2 \\
\Leftrightarrow & \forall\,\sigma_1,\sigma_2,\tau_1,\tau_2 :\; (\,((\sigma_1,\tau_1),(\sigma_2,\tau_2)) \in (p \times I)\,;\rhd\,;\lhd\,;(p \times I)\,) \;\Rightarrow\; \sigma_1 = \sigma_2 \\
\Leftarrow & \forall\,\sigma_1,\sigma_2,\tau_1,\tau_2 :\; (\,((\sigma_1,\tau_1),(\sigma_2,\tau_2)) \in (p \times I)\,;\rhd\,;\lhd\,;(p \times I)\,) \\
& \qquad\qquad \Rightarrow\; ((\sigma_1,\tau_1),(\sigma_2,\tau_2)) \in p \times I \\
\Leftrightarrow & (p \times I)\,;\rhd\,;\lhd\,;(p \times I) \;\subseteq\; p \times I\,.
\end{array}$$

By cancellativity, i.e., for arbitrary $\sigma,\tau_1,\tau_2.\ \sigma \bullet \tau_1 = \sigma \bullet \tau_2 \;\Rightarrow\; \tau_1 = \tau_2$, the above implication turns into an equivalence. □

As a sanity check, we can prove with this result the algebraic $*$-distributivity characterisation used in Definition 3.2.15.

Lemma 4.4.15 *If p satisfies Equation* (4.13) *then for arbitrary tests q, r*

$$p * (q \cap r) = p * q \cap p * r\,.$$

The proof is deferred to Appendix A. We come back to our primary goal to characterise a subset of commands that entail validity of the disjoint decomposition property.

Definition 4.4.16 (Domain Preciseness)
A relation C is called *domain-precise* iff $\ulcorner C$ is precise.

Lemma 4.4.17 *Let C be a domain-precise relation and D any arbitrary relation. Then both relations satisfy Definition 4.4.9.*

Proof. By $(\times/;)$ and neutrality, definition of $*$, property of $\ulcorner$, $(\times/;)$ and neutrality, Lemma 4.4.14, again $(\times/;)$ and neutrality, $(\times/;)$ and $\ulcorner R\,;R = R$ for any relation R,

$$\begin{array}{ll}
& (\ulcorner C \times \ulcorner D)\,;\rhd\,;(C * D) \\
= & (I \times \ulcorner D)\,;(\ulcorner C \times I)\,;\rhd\,;(C * D)
\end{array}$$

$$
\begin{array}{ll}
= & (I \times \ulcorner D) \,;\, (\ulcorner C \times I) \,;\, \rhd \,;\, \lhd \,;\, (C \times D) \,;\, \rhd \\
= & (I \times \ulcorner D) \,;\, (\ulcorner C \times I) \,;\, \rhd \,;\, \lhd \,;\, (\ulcorner C \times I) \,;\, (C \times D) \,;\, \rhd \\
= & (I \times \ulcorner D) \,;\, (\ulcorner C \times I) \,;\, (C \times D) \,;\, \rhd \\
= & (\ulcorner C \times \ulcorner D) \,;\, (C \times D) \,;\, \rhd \\
= & (C \times D) \,;\, \rhd \,.
\end{array}
$$

□

Corollary 4.4.18 *If C is domain-precise then it validates with any other relation D the disjoint concurrency rule for total correctness of Corollary 4.4.11. In particular, all pairs of domain-precise commands validate that inference rule.*

Note, that the reverse direction of Lemma 4.4.17 does not hold as can be seen in Example 4.4.12. The involved commands are not domain-precise since their domain equals $[\![l \mapsto -]\!] * I$. A precise versions of mutation commands can be obtained by setting $[\![[l] := v]\!] = \{(\{(l,n)\}, \{(l,v)\}) \,:\, n \in \mathbb{N}\}$. Another instance of a domain precise command is $\mathsf{dalloc}\,(l) =_{df} \{(\emptyset, \{(l,n)\}) \,:\, n \in \mathbb{N}\}$ that deterministically allocates at address l a new heap cell only for initial empty heaps.

4.4.3 Concurrent Kleene Algebras

A further abstract and general algebraic structure for concurrency is provided by a *Concurrent Kleene Algebra (CKA)* [HMSW11]. A central concept of that algebra is that it allows simple and short soundness proofs of important inference rules like the concurrency and frame rules used in logics for modular reasoning about concurrency. We motivate CKAs in the following by a standard model of it to explain the main basics and central concepts of the algebraic structure. The present form of this model needs to be further refined for incorporating real programs. However, it suffices for our purposes to get an intuition for the abstract concepts. Another concrete model employs predicate transformers to abstractly capture program behaviour of CSL within the setting of CKAs [HHM$^+$11]. It validates a particular part of the CKA laws that already allow a simple soundness proof of the concurrency rule also for that calculus. But unfortunately, that model fails to satisfy other frequently required laws needed for program proofs as, e.g., laws in connection with non-deterministic choice. For any further details we refer to [HHM$^+$11]. Now, the purpose of this section is to investigate the relationally based structure with respect to the laws of a CKA that enable the simple soundness proofs. As a relational structure it also copes well with non-determinacy and moreover allows the re-use of a large and well studied body of algebraic laws in connection with assertion logic. In what follows we largely follow the approach of [DM12a, DM14].

Basically, a starting point for the standard model of [WHO09, HMSW11] is the assumption of a set of events denoted by EV. Concrete examples for this can be simple assignments to variables, a request for shared resources or other communication action between various threads. On the set of events, one defines a *dependence relation* $\rightarrow \subseteq \mathsf{EV} \times \mathsf{EV}$ that is used to express that certain events have to occur before others can be executed. As an example, for modifying a critical resource it first needs to be ensured that the required rights for this are granted. Now a trace tp is defined as a set of events, i.e, $tp \in \mathcal{P}(\mathsf{EV})$ while programs P form sets of traces in that model, i.e., $P \in \mathcal{P}(\mathcal{P}(\mathsf{EV}))$. Moreover, we define that a trace tp is independent of a trace tq by

$$tp \not\leftarrow tq \Leftrightarrow_{df} \neg\exists\, e \in tp, f \in tq : f \rightarrow e\,,$$

i.e., if there are no dependence arrows from events of tp to events of tq. Based on this, one can define various composition operations. We will only enumerate the relevant ones for out purposes. For programs P, Q and $\circ \in \{*\,, ;\,, \mid\}$

$$P \circ Q =_{df} \{tp \cup tq : tp \in P,\, tp \in Q,\, tp\,(\circ)\,tq\}\,,$$

where

$$\begin{array}{l} tp\,(*)\,tq \Leftrightarrow_{df} tp \cap tq = \emptyset\,, \\ tp\,(\,;\,)\,tq \Leftrightarrow_{df} tp \cap tq = \emptyset \wedge tp \not\leftarrow tq\,, \\ tp\,(\mid)\,tq \Leftrightarrow_{df} tp \cap tq = \emptyset \wedge tp \not\leftarrow tq \wedge tq \not\leftarrow tp\,. \end{array}$$

Intuitively, for that model the operation $*$ denotes fine-grained concurrent composition allowing dependencies in both direction while ; denotes sequential composition where P must be independent of Q. Finally, $\mid$ describes disjoint concurrency with no dependencies in any direction.

Next we describe the algebraic structure of this model. Clearly, $(*)$ and $(\mid)$ are symmetric and hence the corresponding operators are commutative. Moreover, the program $\emptyset$ is an annihilator and $\{\emptyset\}$ neutral for all operations. The former can be seen as an erroneous program as in the case of relations while the latter that does nothing and therefore can be interpreted as the program skip. Due to the definitions, one also has the relationship

$$P \mid Q \subseteq P\,;Q \subseteq P * Q$$

for programs P, Q. In [HMSW11] it was shown that $(\mathcal{P}(\mathcal{P}(\mathsf{EV})), \subseteq, *, \{\emptyset\})$ and also $(\mathcal{P}(\mathcal{P}(\mathsf{EV})), \subseteq, ;, \{\emptyset\})$ form quantales (cf. Definition 3.1.1). Moreover, they are connected by the so-called *exchange law* :

$$(P * Q)\,;(R * S) \subseteq (P\,;R) * (Q\,;S) \qquad \text{(exchange)}$$

for programs P, Q, R, S. Intuitively, the program on the left-hand side has more dependencies that require that both programs P and Q has to be executed before R

and S can start while the right-hand side program only states that R needs to be executed after P and S after Q. Hence, S might be executed there before P or R before Q. Both programs of the exchange laws and their dependencies are depicted in Figure 4.4.

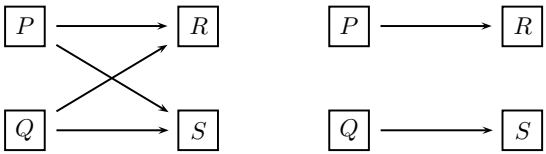


Figure 4.4: Dependencies in the exchange law.

In fact, this inequation is the key ingredient to a simple proof of the concurrency rule in that setting. For presenting that proof we first give a definition of a treatment for Hoare triples within CKAs. For programs P, Q, R, the *general Hoare triple* [HMSW11] is defined as

$$P \,\{Q\}\, R \;\Leftrightarrow_{df}\; P\,;Q \subseteq R\,. \tag{4.14}$$

This states that any legal extension of a P-trace by Q-trace yields a trace of R. Deviating from standard Hoare triples, pre- and postconditions are treated in that setting uniformly as programs. Intuitively, one can think of P as a program that asserts the allocation of the required resources for Q while R abstracts the whole program and only guarantees that the postcondition is satisfied at the end. We will later provide the concrete relationship to standard Hoare triples of the relational approach. In this setting, the concurrency rule has the form

$$\frac{P_1 \,\{Q_1\}\, R_1 \quad P_2 \,\{Q_2\}\, R_2}{P_1 * P_2 \;\{Q_1 * Q_2\}\; R_1 * R_2}\,,$$

where all P_i, Q_i, R_i denote programs. A further essential feature of this rule is that parallel composition is modelled by separation or disjointness $*$ of the involved traces is used to model of programs. The proof is as follows: Assume by (4.14) and the premise of the inference rule $P_i\,;Q_i \subseteq R_i$. Using (exchange) we immediately infer

$$(P_1 * P_2)\,;(Q_1 * Q_2) \;\subseteq\; (P_1\,;Q_1) * (P_2\,;Q_2) \;\subseteq\; Q_1 * Q_2\,.$$

Additionally, it was shown in [HHM$^+$11] that validity of the concurrency rule is equivalent to validity of the exchange law. In the same fashion validity of the frame rule is equivalent to validity of the *small exchange law*

$$(P * Q)\,;R \;\subseteq\; (P\,;R) * Q$$

which follows from (exchange) by setting $S = \{\emptyset\}$ since it is neutral for both operations $*$ and $;$.

For the rest of this section we return to the relational model and denote by $P, Q, \ldots$ relations. Note that by interpreting the operations $*$ and $;$ of the CKA model relationally, we do not have the same neutral element for both operators. Assuming an underlying (single-unit) separation algebra, the neutral element for $*$ is $e =_{df} \{(u,u)\}$ while in the case of $;$ it is given by the identity relation I. Generally, we only have the inclusion $e \subseteq I$ since I is the largest test. If we would assume validity of the exchange laws in our relational model this would immediately imply $I \subseteq e$ by setting $P = R = I$ and $Q = S = e$ in (exchange). By antisymmetry of the order, I and e would be equal, a contradiction. Therefore, the relational exchange law cannot be valid. For a more concrete analysis of this problem, we provide a simple example with commands defined on the heap model in (4.11). We will provide for the rest of this section all examples on this particular separation algebra.

Example 4.4.19 A concrete counterexample of the exchange law can be given by setting $P = [\![1 \mapsto 2]\!]$, $Q = [\![2 \mapsto 3]\!]$, $R = [\![[2] := 4]\!]$, $S = [\![[1] := 5]\!]$ in (exchange). By definition we have $P * Q = \{(\{(1,2),(2,3)\},\{(1,2),(2,3)\})\}$ and $(\{(1,2),(2,3)\},\{(1,5),(2,4)\}) \in R * S$. Hence the left-hand side of the exchange law is non-empty. Now, the right-hand side of the law resolves to

$$P\,;R = [\![1 \mapsto 2]\!]\,;[\![[2] := 4]\!] = \emptyset = Q\,;S = [\![2 \mapsto 3]\!]\,;[\![[1] := 5]\!]\,.$$

Therefore the composed relation equals the empty set and thus the rule is violated. □

Note that in the above example, although $P * Q$ provides the required set of resources for $R * S$, the refined program coincides with divergence. It can be seen that the heap cells at addresses 1 and 2 are distributed to the wrong commands, respectively. This is due to the angelic semantics of the inclusion order on relation. Hence, an idea to validate the exchange law with relations could be to use a different ordering. In fact, and surprisingly, it is possible to show validity of a restricted variant of the exchange law with the reversed inclusion order. This gives the behaviour of the order a demonic flavour. The proof for this uses a restriction on pairs (P, Q) of relations: when P and Q start from combinable pairs of input states they will produce combinable pairs of output states, or the other way around. This is formalised as follows.

Definition 4.4.20 We define that relations P and Q are *forward compatible* iff

$$\#\,;(P \times Q) \;\subseteq\; (P \times Q)\,;\#\,.$$

Symmetrically P and Q are *backward compatible* iff $(P \times Q)\,;\# \subseteq \#\,;(P \times Q)$. Two relations are called *compatible* iff they are forward and backward compatible, i.e., $\#\,;(P \times Q) = (P \times Q)\,;\#$.

Example 4.4.21 For an intuition of the concept of forward compatible commands we additionally define for the separation algebra of heaps the relational denotation

$$[\![\, \mathsf{dalloc}\,(l)\,]\!] \;=_{df}\; \{(h, \{(l,n)\} \cup h) : x \notin dom(h),\, n \in \mathbb{N}\} \tag{4.15}$$

that allocates a fresh heap cell at address l with arbitrary contents. Now, consider heaps $h_1 = \{(1,1)\}$, $h_2 = \emptyset$ and $h_1' = h_2' = \{(1,2)\}$. Recall that $\#$ holds if the considered heaps have disjoint domains. Clearly, $h_1 \# h_2$ and $((h_1,h_2),(h_1',h_2')) \in \#\,;([\![\,[1] := 2\,]\!] \times [\![\,\mathsf{dalloc}\,(1)\,]\!])$. But $h_1' \# h_2'$ does not hold and thus $((h_1,h_2),(h_1',h_2')) \notin ([\![\,[1] := 2\,]\!] \times [\![\,\mathsf{dalloc}\,(1)\,]\!])\,;\#$.

By changing $\mathsf{dalloc}\,(1)$ to $\mathsf{dalloc}\,(l)$ for $l \neq 1$, one would end up with compatible commands. In that case, the two compatible commands would work on disjoint portions of the heap and hence ensure disjointness before and after their execution. Such a condition needs to ensured e.g., when a resource context is considered in the environment. □

The notion of compatibility entails some useful properties that we list below. For better readability, a few proofs and auxiliary results have been moved to the Appendix A.

Lemma 4.4.22 *Assume P, Q are forward compatible. Then $\ulcorner P * \ulcorner Q = \ulcorner(P * Q)$, i.e., $*$ distributes over domain.*

A proof can be found in the Appendix. Note that without the concept of compatibility, it was only possible to show an inclusion (cf. Lemma 4.3.10).

Lemma 4.4.23 *All tests are compatible with each other. In particular, I is compatible with itself.*

Proof. For test p, q the relation $p \times q$ is a test in the algebra of relations on pairs. Since $\#$ is a test there, too, they commute, which means forward and backward compatibility of p and q. □

Finally, by the use of forward or backward compatible relations we are able to prove validity of the exchange law using the reversed inclusion order.

Theorem 4.4.24 (Reverse Exchange) *If P, Q are forward compatible or R, S are backward compatible then*

$$(P\,;R) * (Q\,;S) \;\subseteq\; (P * Q)\,;(R * S)\,.$$

In particular, if P, Q or R, S are tests the inequation holds.

Proof. We assume that P and Q are forward compatible. By definition of $*$, $(\times/;)$, Lemma 4.3.4, forward compatibility, Lemma 4.3.4, and definition of $*$:

$$\begin{array}{ll} & (P\,;R) * (Q\,;S) \\ = & \lhd\,;(P\,;R\times Q\,;S)\,;\rhd \\ = & \lhd\,;(P\times Q)\,;(R\times S)\,;\rhd \\ = & \lhd\,;\#\,;(P\times Q)\,;(R\times S)\,;\rhd \\ \subseteq & \lhd\,;(P\times Q)\,;\#\,;(R\times S)\,;\rhd \\ \subseteq & \lhd\,;(P\times Q)\,;\rhd\,;\lhd\,;(R\times S)\,;\rhd \\ = & (P*Q)\,;(R*S)\,. \end{array}$$

The proof for backward compatibility and R, S is symmetric. □

The reverse exchange law expresses an increase in granularity: while in the left-hand side programs $P\,;R$ and $Q\,;S$ are treated as indivisible, they are split in the right-hand side program, at the expense of a "global" synchronisation point marked by the semicolon (cf. Figure 4.5).

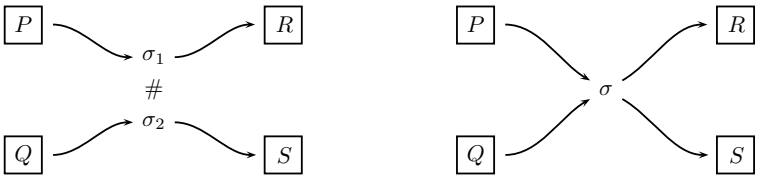


Figure 4.5: Compatibility in the reverse exchange law.

The possibility of such a synchronisation point is established on the left-hand side by the compatibility requirement, i.e., in Figure 4.5, the output states σ_1 of P and σ_2 of Q allow their combination into a global state σ, as in the right-hand side program. In other words, the implicit split in the left-hand side is one of the possible splits admitted by $\rhd\,;\lhd$ in the right-hand side. Still another way of viewing the rule is that the right-hand side "forgets" information about splits and therefore is more liberal.

Example 4.4.25 As an example, we can define programs $\mathit{produce}(l)$ to produce some resource at address l and $\mathit{consume}(l)$ for consuming the corresponding resource at address l. The programs can be relationally realised e.g., by $[\![\,\mathsf{dalloc}\,(l)\,]\!]$ from Equation (4.15) and $[\![\,\mathsf{delete}\,(l)\,]\!] =_{df} \{(h, h-\{(l,n)\}) : (l,n)\in h,\, n\in\mathbb{N}\}$. Next, consider for an $l\in\mathbb{N}$ the composed program

$$(\mathit{produce}(l)\,;\mathit{consume}(l)) * (\mathit{produce}(l+1)\,;\mathit{consume}(l+1))\,.$$

In this program, each producer and its corresponding consumer are treated together as an indivisible program. Since the producers and consumers work on disjoint resources, they are compatible and we can use the reverse exchange law to reorder the program above into

$$(\mathit{produce}(l) * \mathit{produce}(l+1)) \,;\, (\mathit{consume}(l) * \mathit{consume}(l+1)) \,.$$

This version represents a program where all resource allocations need to be executed concurrently before any of the resources can be consumed. The synchronisation point denoted by ; reflects an intermediate state that includes the produced resources at the addresses l and $l+1$. □

As a further step we connect the definition of generalised Hoare triples (4.14) that admit programs as assertions with a well-known one for Hoare logic (cf. Equation (4.2)), i.e., $p \,;\, Q \subseteq Q \,;\, r$ for any relation Q and tests p, r. The major difference of both definitions is that the former approach does not impose any structural restriction on the denotations for modelling pre- and postconditions.

As shown in Lemma 4.3.26, we have the relationship

$$p \,;\, Q \subseteq Q \,;\, r \;\Leftrightarrow\; \top \,;\, p \,;\, Q \subseteq \top \,;\, r \;\Leftrightarrow\; (\top \,;\, p)\,\{\!|Q|\!\}\,(\top \,;\, r) \,.$$

This allows in particular to immediately translate results for standard Hoare triples into ones for general triples. The composition $\top \,;\, p$ maps a test p to a more general relation that makes no assumption about its initial starting states, i.e., any execution from an arbitrary state will end up in one contained in p. Trivially, the symmetric relation $p \,;\, \top$ makes no restriction on final states or its codomain.

Unfortunately, the modified exchange law introduces an inconsistency, since the order for the proved exchange rule is reversed in contrast to the order of the relational interpretation for general Hoare triples. However, one can obtain some further results by this, which from a theoretical point of view might help for future considerations about the exchange law and relational models.

As a first approach we note that for a relational treatment of faulting within a separation logic, we added in the total correctness approach to the semantics of Hoare triples (cf. Definition 4.2.10) an enabledness condition $p \subseteq \lceil Q$ as additional conjunct. We will see that reversing this condition, i.e., stating $\lceil Q \subseteq p$ entails a soundness proof of a variant of the concurrency rule using the reverse exchange law. The condition states that Q enforces the precondition p in that all of its initial states need to satisfy p. With this we define for tests p, r and relation Q

$$\{\!|p|\!\}\,Q\,\{\!|r|\!\} \;\Leftrightarrow_{df}\; p \,;\, Q \subseteq Q \,;\, r \wedge \lceil Q \subseteq p \,. \tag{4.16}$$

This yields some useful properties and conditions. The following observation is trivial, but useful for our first variant of the concurrency rule.

Lemma 4.4.26 $\{p\,;q\}\,Q\,\{r\} \;\Leftrightarrow\; \{p\}\,q\,;Q\,\{r\} \wedge \lceil Q \subseteq p\,.$

Proof. By definition and associativity we immediately infer $(p\,;q)\,;Q \subseteq Q\,;r \;\Leftrightarrow\; p\,;(q\,;Q) \subseteq Q\,;r\,.$ Moreover, $\lceil Q \subseteq p\,;q \subseteq p$ and $\lceil Q = \lceil(\lceil Q\,;Q) \subseteq \lceil(q\,;Q) \subseteq p\,.$ □

This lemma specialises in a number of ways. Since tests are idempotent we obtain by setting $q = p$ that

Corollary 4.4.27 $\{p\}\,Q\,\{r\} \;\Leftrightarrow\; \{p\}\,p\,;Q\,\{r\} \wedge \lceil Q \subseteq p\,.$

The relation $p\,;Q$ can be viewed as an execution that first asserts the precondition p before executing Q. Next, we may set $p = I = [\![\,\mathsf{true}\,]\!]$ in Lemma 4.4.26 to get

Corollary 4.4.28 $\{q\}\,Q\,\{r\} \;\Leftrightarrow\; \{\mathsf{true}\}\,q\,;Q\,\{r\}\,.$

Note that the condition $\lceil Q \subseteq p$ of the involved triples is equivalent to the formula $Q \subseteq p\,;Q$. This can be further strengthened to an equation, i.e., $Q = p\,;Q$. Hence, we can conclude

Theorem 4.4.29 *For tests p_1, p_2, r_1, r_2 and relations Q_1, Q_2 we have*

$$\frac{\{p_1\}\,Q_1\,\{r_1\} \quad \{p_2\}\,Q_2\,\{r_2\}}{\{\mathsf{true}\}\,Q_1 * Q_2\,\{\,r_1 * r_2\}}\,.$$

Proof. By $[\![\,\mathsf{true}\,]\!] = I$, $\{p_i\}\,Q_i\,\{r_i\}$ implies $Q_i \subseteq p_i;Q_i$, by $\{p_i\}\,Q_i\,\{r_i\}$, and reverse exchange law (Lemma 4.4.24) with tests r_1 and r_2:

$$I;(Q_1{*}Q_2) \;=\; Q_1{*}Q_2 \;\subseteq\; (p_1;Q_1){*}(p_2;Q_2) \;\subseteq\; (Q_1;r_1){*}(Q_2;r_2) \;\subseteq\; (Q_1{*}Q_2);(r_1{*}r_2)\,.$$

□

Note that compatibility of the commands Q_i is not needed for instantiating the reverse exchange law. Hence, the compatibility requirement is not a restriction for this application. In particular, Theorem 4.4.29 states that these inference rules are very liberal w.r.t. the involved preconditions, also in combination with disjoint concurrent compositions. The requirement by this is that all executions of each relation Q_i have to be enabled by the precondition. We can bring this inference rule into a form closer to the original and more common version:

Corollary 4.4.30 *Theorem 4.4.29 is equivalent to*

$$\frac{\{p_1\}\,Q_1\,\{r_1\} \quad \{p_2\}\,Q_2\,\{r_2\}}{\{p_1 * p_2\}\,Q_1 * Q_2\,\{\,r_1 * r_2\}}\,.$$

Proof. By the second proof step above, reverse exchange and $p_1 * p_2 \subseteq I$ we have

$$Q_1 * Q_2 \subseteq (p_1 ; Q_1) * (p_2 ; Q_2) \subseteq (p_1 * p_2) ; (Q_1 * Q_2) \subseteq Q_1 * Q_2 .$$

Hence $Q_1 * Q_2 = (p_1 * p_2) ; (Q_1 * Q_2)$, so that Corollary 4.4.28 shows the claim. □

The following result provides together with Theorem 4.4.29 the analogue of the equivalence between the exchange law and the concurrency rule as shown in [HHM+11] also for the relation-based treatment.

Lemma 4.4.31 *Validity of Theorem 4.4.29 implies a special case of the reverse exchange law: for arbitrary commands P_i and tests r_i,*

$$(P_1 ; r_1) * (P_2 ; r_2) \subseteq (P_1 * P_2) ; (r_1 * r_2) .$$

The proof can be found in Appendix A.

We remark that the definition of the triples provided in Equation (4.16) unfortunately does not validate unrestricted strengthening of the precondition p as it has to include at least the domain of the relation Q. Therefore, we continue with another possibility to interpret the reversed order of the exchange law that allows the mentioned weakening or strengthening of proof rules. The idea is to link that law with a dual definition of triples w.r.t. one of standard Hoare triples. By this we will again see that the concurrency and frame rules for those triples can easily be derived using the reverse exchange law. First, we give a definition of [Hoa11].

Definition 4.4.32
For relations P, Q, R we define *Plotkin* triples by

$$\langle P, Q\rangle \to R \;\; \Leftrightarrow_{df} \;\; R \subseteq P ; Q$$

and *dual partial correctness* triples by

$$P \,[Q]\, R \;\; \Leftrightarrow_{df} \;\; P \subseteq Q ; R .$$

Note that in comparison with generalised Hoare triples the ;-composed relation is on right-hand side of the inclusion order. Intuitively, the former characterises a set of possible final states satisfying the postcondition R after execution of Q from initial states of the precondition P. For this one can think of labelled transition systems, where Q represents some sequence of actions that possibly leads from a configuration or state in P to some final configuration of R. The semantics of such triples is rather angelic since it states only that such a transition may exists. The notation is inspired by Plotkin's structural operational semantics [Plo04] in which $\langle s, C\rangle \to t$ means that evaluation of term C starting in state s may lead to a term t.

The dual partial correctness triples describe possible starting states of P that end in R after some execution of Q. According to [Hoa11], dual partial correctness triples can, e.g., be used as a method for the generation of test cases. Assuming that R represents erroneous final states of Q, then P characterises some conditions that will lead to such error situations. Plotkin triples can be used for a dual application.

Using again the relationship of Lemma 4.3.26 the dual partial correctness triples transform for tests p, q into

$$(p\,;\top)\,[Q]\,(q\,;\top) \;\Leftrightarrow\; p\,;\top \subseteq Q\,;(q\,;\top) \;\Leftrightarrow\; p \subseteq (Q\,;q)\,;\top \;\Leftrightarrow\; p \subseteq \lceil (Q\,;q)$$

and, symmetrically, Plotkin triples transform into

$$\langle \top\,;p\,,Q\rangle \to \top\,;q \;\Leftrightarrow\; \top\,;q \subseteq (\top\,;p)\,;Q \;\Leftrightarrow\; q \subseteq \top\,;(p\,;Q) \;\Leftrightarrow\; q \subseteq (p\,;Q)^{\urcorner}\,,$$

where $\_^{\urcorner}$ denotes the symmetric codomain operator. We concentrate on dual partial correctness triples and use the abbreviation

$$p\,[[Q]]\,q \;\Leftrightarrow_{df}\; (p\,;\top)\,[Q]\,(q\,;\top) \;\Leftrightarrow\; p \subseteq \lceil (Q\,;q)\,. \tag{4.17}$$

The dual results can be similarly calculated for Plotkin triples. Since the order within these triple definitions works in the same direction as in the reverse exchange laws we can immediately state the following result.

Lemma 4.4.33 *The concurrency rule for dual partial correctness or Plotkin triples holds iff the reverse exchange law holds.*

A proof for this lemma can be derived dually to the proof of [HHM+11] stating that the exchange law is equivalent to the concurrency rule involving general Hoare triples. Unfortunately, in our setting the reverse exchange law does only hold conditionally w.r.t. the assumption of compatible pairs of relations. In contrast to the approach that used the triple definition of Equation (4.16), the compatibility assumption is required for a soundness proof of the concurrency rule involving the triples of Equation (4.17).

Theorem 4.4.34 *If Q_1, Q_2 are forward compatible then the concurrency rule for dual partial correctness triples holds, i.e., for tests p_1, p_2, q_1, q_2*

$$\frac{p_1\,[[Q_1]]\,q_1 \qquad p_2\,[[Q_2]]\,q_2}{p_1 * p_2\,[[Q_1 * Q_2]]\,q_1 * q_2}\,.$$

Again this is also valid when Q_1 and Q_2 are backward compatible and Plotkin instead of dual partial correctness triples are used.

Proof. By assumption we have $p_1 \subseteq \lceil(Q_1;q_1)$, $p_2 \subseteq \lceil(Q_2;q_2)$ and the restricted variant of the reverse exchange law. Hence, by isotony and Lemma 4.4.22 we calculate

$$p_1 * p_2 \subseteq \lceil(Q_1 ; q_1) * \lceil(Q_2 ; q_2) = \lceil((Q_1 ; q_1) * (Q_2 ; q_2)) \subseteq \lceil((Q_1 * Q_2) ; (q_1 * q_2)) .$$

□

Note that an advantage of this relational treatment is that it allows the simple usage of tests for modelling pre- and postconditions while the standard model of CKAs only has a trivial test algebra consisting of the elements $\emptyset$ and $\{\emptyset\}$.

As a further result, it was shown in [HHM+11] that using general Hoare triples, the frame rule is equivalent to the small exchange law, i.e., $(P * Q) ; R \subseteq (P ; R) * Q$ for programs P, Q, R. This form could be obtained from the exchange law if $*$ and ; would have the same unit. This is not the case for relations since we generally only have one in equation (cf. Lemma 4.3.30). If we would assume similar as in [HHM+11] relations satisfying the compact characterisation of Lemma 4.3.31 we can also conclude

Corollary 4.4.35 *If $Q * I = Q$ and Q is forward compatible with I then the frame rule for dual partial correctness triples holds, i.e., for tests p, q, r*

$$\frac{p\,[[Q]]\,q}{p * r\,[[Q]]\,q * r} .$$

(A dual result again holds for Plotkin triples).

The behaviour of the triples of Definition 4.4.32 is called "dual" on purpose, since the calculations given above are symmetric to the algebraic approach of [HHM+11]. Both provided inference rules have the restriction that compatible pairs of involved relations are needed. The reason for this is that, by Lemma 4.3.4 only $\# \subseteq \rhd ; \lhd$ is valid. The restriction could be excluded if $I \times I \subseteq \rhd ; \lhd$ could be established. In particular, this would yield non-restricted validity of the reverse exchange law. However this requires an artificial extension of the split and join relations to total ones, i.e., $\rhd ; \lhd$ needs to include an extra state as a result for non-combinable pairs of states. This can be done by enriching the underlying separation algebra by a fresh state $\sigma_\lightning$ and using a new combinator $\circ$, instead of $\bullet$, defined by

$$\sigma \circ \tau = \sigma_\lightning \quad \Leftrightarrow_{df} \quad \neg\, \sigma \# \tau$$

and $\sigma \circ \tau = \sigma \bullet \tau$ in any other case. The extended carrier set can then be given by $\Sigma_\lightning \ =_{df} \ \Sigma \,\dot\cup\, \{\sigma_\lightning\}$. For theoretical considerations we can construct by this a relational model that allows a connection to the algebraic structure of a *locality bi-monoid* [HHM+11]. Its general purpose is to connect behaviour of CSL with the structure of a CKA.

Definition 4.4.36
A *locality bimonoid* is defined by $(S, \leq, *, 1_*, ;, 1_;)$ where $(S, \leq)$ is partially ordered and the operations $*$, $;$ are monotone w.r.t. the carrier set S. Moreover, $(S, *, 1_*)$ needs to form a commutative monoid and $(S, ;, 1_;)$ a monoid. Additionally, the structure has to satisfy the exchange law and $1_; * 1_; = 1_;$.

By the use of the extended split and join relations, we can interpret $\leq$ as the reverse set inclusion order $\supseteq$ since the reverse exchange law holds unconditionally. In summary, using Lemma 4.3.6, we have the following result.

Lemma 4.4.37 $(\mathcal{P}(\Sigma_\lightning \times \Sigma_\lightning), \supseteq, *, e, ;, I)$ *forms a locality bimonoid.*

These considerations are rather of theoretical interest. Note that by reversing the set inclusion order turns $\sqcup$ and $\sqcap$ into $\cap$ and $\cup$. By using the test subalgebra as algebraic counterpart to model logical assertions the interpretation of the notion of a test becomes unnatural. The reason for this is that $p \wedge q$ will be identified, unusually, with $p \sqcup q$ and $p \vee q$ with $p \sqcap q$. From an algebraic viewpoint these modifications entail simplifications, since no additional constraints are required to validate the reverse exchange law. However, by considering the extra failure-state $\sigma_\lightning$ one deviates from the usual angelic semantics of relations.

4.5 Pointfree Dynamic Frames

In this section we present another application for the derived algebraic abstractions of the principles and concepts of separation logic. Besides allowing compact reasoning about shared mutable data structures it also represents by its frame rule an adequate solution to the *frame problem* [MH69]. Concretely, the frame problem asks for a methodology that allows specifying which resources of a program can be changed and which ones are left unchanged without naming them explicitly. Such a methodology should additionally guarantee modularity and hence scalability in specification and correctness proofs of programs. A further popular approach to the frame problem is the *theory of dynamic frames* [Kas11] that provides the mentioned modularity while still being expressive enough to handle a variety of concrete programs. There exist further variations of the theory that address the automation of program verification (e.g., [SJP09, Lei10, GGN11]).

Now, the main goal of the following considerations is to develop algebraic abstractions for the theory of dynamic frames similar as we have provided for separation logic. In particular, by including the former approach into the extended relational structure, a more general treatment of resources within separation algebras (cf. Section 3.3) is

possible. Moreover, pointfree characterisations of central properties will allow simple proofs of crucial concepts in a calculational style. Generally, the provided relational calculus for separation logic further extends towards a unifying approach including also the theory of dynamic frames. In what follows we basically follow [Dan14].

4.5.1 Abstracting Dynamic Frames

The basic setting for resource states in the theory of dynamic frames are finite mappings from an infinite set of locations Loc to an infinite set of values Val that comprises at least integers and Booleans. This closely corresponds to the (single-unit) separation algebra based on simple heaps (cf. Example 3.3.2 a)). Hence, we have a concrete instance of a separation algebra for which we formally write $\mathsf{DFSA} =_{df} (\mathsf{Loc} \rightsquigarrow \mathsf{Val}, \dot\cup, \emptyset)$ where $\dot\cup$ denotes union of location-disjoint functions, $\emptyset$ the completely undefined function and $\sigma \# \tau \Leftrightarrow dom(\sigma) \cap dom(\tau) = \emptyset$. We write $dom(\sigma)$ for a mapping or state σ to denote its domain or more concretely all of its allocated locations, i.e., a subset of Loc. Moreover, we use DFSA as an abbreviation for the dynamic frames separation algebra and define the substate $\sigma\big|_X$ that restricts the domain of the state σ to a set of locations X.

Lemma 4.5.1 *For a state τ assume $dom(\tau) = X$. Then for arbitrary σ we have $(\sigma \bullet \tau)\big|_X = \tau$.*

We concentrate for this section only on single-unit separation algebras as we do not require multi-unit ones. It is not a difficult task to extend the treatment to separation algebras involving multi-units. As a next step, we provide abstractions for the concrete dynamic frames resource setting and manage several central properties of the approach within the abstraction to separation algebras. For this we require additional assumptions given in [DHA09] and basically follow the approach of that work. A separation algebra $(\Sigma, \bullet, u)$ satisfies *disjointness* iff for all σ, τ

$$\sigma \bullet \sigma = \tau \;\Rightarrow\; \sigma = \tau \tag{4.18}$$

and it satisfies *cross-split* iff for arbitrary states σ_i with $i \in \{1,2,3,4\}$

$$\begin{aligned} \sigma_1 \bullet \sigma_2 = \sigma_3 \bullet \sigma_4 \;\Rightarrow\; \exists\, \sigma_{13}, \sigma_{14}, \sigma_{23}, \sigma_{24} :\; & \sigma_1 = \sigma_{13} \bullet \sigma_{14} \;\wedge\; \sigma_2 = \sigma_{23} \bullet \sigma_{24} \\ & \wedge\; \sigma_3 = \sigma_{13} \bullet \sigma_{23} \;\wedge\; \sigma_4 = \sigma_{14} \bullet \sigma_{24}\,. \end{aligned} \tag{4.19}$$

Disjointness in the presence of cancellativity implies that the only element that can be combined with itself is the neutral element u, i.e.,

$$\sigma \# \sigma \;\Rightarrow\; \sigma = u\,. \tag{4.20}$$

Equivalently, non-unit elements cannot be combined with themselves since any allocated resources will overlap in such products. Therefore, the condition of (4.18) is called disjointness. For a proof of (4.20) assume a state σ that satisfies $\sigma \# \sigma$. By definition of $\#$, Equation (4.18), and a logic step, we have:

$$\sigma \# \sigma \;\Leftrightarrow\; (\exists\, \tau :\; \sigma \bullet \sigma = \tau) \;\Rightarrow\; (\tau = \sigma) \;\Rightarrow\; (\sigma \bullet \sigma = \sigma)\,.$$

Now, by cancellativity we can infer $u \bullet \sigma = \sigma \bullet \sigma \;\Rightarrow\; u = \sigma$.

The idea of the cross-split assumption can be explained as follows: assume that a state can be combined in two ways or that there exist two possible splits of a state. Then there need to exist four substates that represent a partition of the original state w.r.t. the mentioned splits. The partitions of the state can be depicted as follows:

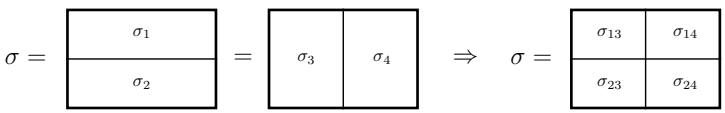


Figure 4.6: Illustration of the cross-split assumption for a state σ.

For the remaining sections we assume separation algebras that satisfy disjointness and cross-split. A concrete example of such a separation algebra can be found in [HV13]. The assumptions are required there to establish basic properties of operators for reasoning about sharing within data structures. Note that the separation algebra DFSA also satisfies disjointness and cross-split.

Dynamic frames are represented in concrete program specifications as specification variables, i.e., variables that serve only for verification purposes and hence are not physically visible in the program itself. Their usage is to cover a set of locations, also called a region, of a state σ ranging over variables or allocated objects. By this mechanism one obtains the expressiveness to specify what a program or a method is allowed to modify and what remains untouched during its execution. Frequently used examples of the theory of dynamic frames are the auxiliary specification variables

$$\mathsf{used} = \mathsf{used}_\sigma \;=_{df}\; dom(\sigma) \qquad \text{and} \qquad \mathsf{unused} \;=_{df}\; \mathsf{Loc} - \mathsf{used}\,.$$

The former denotes the set of locations to which the state σ assigns values while the latter corresponds to all unallocated ones in that state. A *dynamic frame* f at a state σ is defined as a subset of Loc satisfying $f \subseteq \mathsf{used}$. Hence, dynamic frames are state dependent and may vary with state transitions. Following the notation in [Kas11] a dynamic frame f in a final state σ' is denoted by f', i.e., it would correspond to $f_{\sigma'}$.

module *Rat*
spec var $rat\_inv \in \mathbb{B}, rat, rep$
$rat\_inv \;\Rightarrow\; rat \in \mathbb{Q}$
$rat\_inv \;\Rightarrow\; rat\_rep \subseteq \mathsf{used} \wedge rat\_rep$ **frames** (rat_inv, rat)
proc $double() : \; rat\_inv \;\Rightarrow\; rat' = 2 \cdot rat \wedge rat\_inv'$
end module

Figure 4.7: Specification of a rational number module with dynamic frames.

As a concrete example given in Figure 4.7, we present a rational number module of [Kas11]. In that example rat_inv is a Boolean specification variable and abstractly specifies the invariant of that module requiring in particular that rat has to be a rational number. The specification variables rep and rat_rep are dynamic frames where the former belongs to the module and the latter to rational number itself. By the keyword **frames** it is asserted that rat_rep covers all the locations of the variables of rat_inv and rat. This mechanism is introduced to ensure that all variables framed by rat_rep will remain unchanged as long as rat_rep is not changed, e.g., by any other procedure that uses an implementation of the module *Rat*. Therefore, the dynamic frames are also called representation regions. The specification of the procedure $double()$ states that whenever the invariant of the rational number holds then the procedure asserts after its execution that the final value of rat equals the doubled initial value and the invariant of rat still holds.

For an abstraction of the theory of dynamic frames we start our considerations from the concrete separation algebra DFSA. Since the theory of dynamic frames does not need to distinguish program abortion from non-termination we will use the simpler relational structures on the carrier set $\Sigma \times \Sigma$. For a relational treatment we use a constant sets of locations to represent initial dynamic frames f. The dynamic behaviour within state transitions $\sigma\, Q\, \sigma'$ for a relation Q will be represented by relational and pointfree formalisations rather than using functions or expressions that depend on the states σ or σ' as in the original approach. This will allow more concise structural characterisations and pointfree proofs of basic properties involving dynamic frames.

More concretely, assuming an initial dynamic frame f to be a fixed set of locations we define

$$[\![f]\!] =_{df} \{(\sigma, \sigma) : dom(\sigma) = f\},$$

i.e., embedding f as a relation yields a subidentity or a test which characterises all states where the allocated set of locations equals f. Note that $[\![f]\!] \neq \emptyset$, even if $f = \emptyset$, because then $[\![f]\!] = \{(u, u)\}$. For better readability we will omit the $[\![\_]\!]$ brackets in the following. The context will disambiguate the usage. This embedding of f implies that the corresponding test satisfies the special behaviour of *precise* tests

for which we use the pointfree characterisation of Lemma 4.4.14, i.e.,

$$(f \times I)\,;\rhd\,;\lhd\,;(f \times I) \subseteq f \times I.$$

This means that in any state τ a unique substate w.r.t. $\preceq$ that contains exactly the locations of f can always be pointed out.

As the next step we introduce pointfree relational variants of *framing requirements* that are crucial for the theory of dynamic frames [Kas11].

Definition 4.5.2 (Framing Requirements) Assume a dynamic frame f. Then the *modification* Δf and *preservation* Ξf are defined relationally by

$$\begin{aligned}\Delta f &=_{df} \{(\sigma,\sigma') : \sigma\big|_{\mathsf{used}-f} = \sigma'\big|_{\mathsf{used}-f}\}\,,\\ \Xi f &=_{df} \{(\sigma,\sigma') : \sigma\big|_{f} = \sigma'\big|_{f}\}\,.\end{aligned}$$

The modification requirement Δf intuitively asserts that at most resources captured by the frame f can be changed while any other resources remain untouched and hence are not modified. In particular, Δf allows the allocation of fresh storage. Conversely, Ξf asserts that at least the state parts characterised by f are not changed while anything else can be changed arbitrarily.

Theorem 4.5.3 *Assume a dynamic frame f. Then*

$$\Delta f = (f\,;\top) * I \qquad \textit{and} \qquad \Xi f = f * \top\,.$$

Proof. By definition of $\Delta\_$, definition of I, by set theory and definition of $\top$, using f is a test, definition of $;$, and definition of $*$:

$$\begin{aligned}&\sigma\ (\Delta f)\ \sigma'\\ \Leftrightarrow\ &\sigma\big|_{\mathsf{used}-f} = \sigma'\big|_{\mathsf{used}-f}\\ \Leftrightarrow\ &\sigma\big|_{\mathsf{used}-f}\ I\ \sigma'\big|_{\mathsf{used}-f}\\ \Leftrightarrow\ &\sigma\big|_{\mathsf{used}-f}\ I\ \sigma'\big|_{\mathsf{used}-f} \wedge \sigma\big|_{f}\ \top\ \sigma'\big|_{\mathsf{used}'-(\mathsf{used}-f)}\\ \Leftrightarrow\ &\sigma\big|_{\mathsf{used}-f}\ I\ \sigma'\big|_{\mathsf{used}-f} \wedge \sigma\big|_{f}\ \top\ \sigma'\big|_{\mathsf{used}'-(\mathsf{used}-f)} \wedge \sigma\big|_{f}\ f\ \sigma\big|_{f}\\ \Leftrightarrow\ &\sigma\big|_{\mathsf{used}-f}\ I\ \sigma'\big|_{\mathsf{used}-f} \wedge \sigma\big|_{f}\ (f\,;\top)\ \sigma'\big|_{\mathsf{used}'-(\mathsf{used}-f)}\\ \Rightarrow\ &\sigma\ ((f\,;\top) * I)\ \sigma'\,.\end{aligned}$$

For the reverse implication assume states $\sigma_f, \sigma_I, \sigma_\top$ with $\sigma_f \in f \wedge \sigma = \sigma_f \bullet \sigma_I \wedge \sigma' = \sigma_\top \bullet \sigma_I$. Using Lemma 4.5.1 we get $\sigma\big|_f = (\sigma_f \bullet \sigma_I)\big|_f = \sigma_f$. Hence, $\sigma = \sigma\big|_f \bullet \sigma\big|_{\mathsf{used}-f}$ and cancellativity implies $\sigma_I = \sigma\big|_{\mathsf{used}-f}$. Moreover, we can infer $\sigma'\big|_{\mathsf{used}-f} = (\sigma_\top \bullet$

$\sigma_I)\big|_{\mathsf{used}-f} = (\sigma_\top \bullet \sigma\big|_{\mathsf{used}-f})\big|_{\mathsf{used}-f} = \sigma\big|_{\mathsf{used}-f}$. Now Lemma 4.5.1 implies $\sigma_\top = \sigma'\big|_{\mathsf{used}'-(\mathsf{used}-f)}$.

By definition of $\Xi\_$, f being a test, set theory, definition of $\top$ and definition of $*$:

$$\begin{array}{ll} & \sigma\ (\Xi f)\ \sigma' \\ \Leftrightarrow & \sigma\big|_f = \sigma'\big|_f \\ \Leftrightarrow & \sigma\big|_f\ f\ \sigma'\big|_f \\ \Leftrightarrow & \sigma\big|_f\ f\ \sigma'\big|_f \wedge \sigma\big|_{\mathsf{used}-f}\ \top\ \sigma'\big|_{\mathsf{used}'-f} \\ \Rightarrow & \sigma\ (f * \top)\ \sigma' . \end{array}$$

The reverse implication can be proved analogously to the above case. □

Hence the framing requirements can be completely described within the $*$-extended relational structure. Moreover, the algebraic embedding of dynamic frames as precise tests and their use in pointfree characterisations of the framing requirements yield the abstraction from the concrete DFSA separation algebra to arbitrary ones. We will use the terms dynamic frame and precise test in the following as synonyms. The abstraction further allows calculational proofs of fundamental properties that establish the theory as a solution to tackle the frame problem. We begin with the following result: Assume two initial disjoint sets of locations f, g where at most those of f can be modified. By this all locations of g will remain unchanged. The general idea of this is that expressions depending on locations of f will not affect expressions that depend only on locations in g. An abstraction of this fact is stated in the following result.

Lemma 4.5.4 *Assume dynamic frames f, g. Then*

$$(f * g * I)\,;\Delta f \subseteq g * \Delta f .$$

Proof. By Theorem 4.5.3, definition of $*$, neutrality of I and exchange $(\times/;)$, f is precise (Equation (4.13)), I is neutral and exchange $(\times/;)$, definition of $*$, and commutativity of $*$ and Theorem 4.5.3:

$$\begin{array}{ll} & (f * g * I)\,;\Delta f \\ = & (f * g * I)\,;((f\,;\top) * I) \\ = & \lhd\,;(f \times (g * I))\,;\rhd\,;\lhd\,;(f\,;\top \times I)\,;\rhd \\ = & \lhd\,;(I \times (g * I))\,;(f \times I)\,;\rhd\,;\lhd\,;(f \times I)\,;(\top \times I)\,;\rhd \\ \subseteq & \lhd\,;(I \times (g * I))\,;(f \times I)\,;(\top \times I)\,;\rhd \\ = & \lhd\,;(f\,;\top \times (g * I))\,;\rhd \\ = & (f\,;\top) * g * I \\ = & g * \Delta f . \end{array}$$

□

Since a dynamic frame f covers a set of locations in a state, it can be concluded that as long as f is not changed all variables and expressions that depend on its locations will also remain unchanged. Expressions E can be abstracted relationally to tests that only include the states that assign values to at least all free variables occurring in E. Abstractly, we define that a dynamic frame f *frames a test* E iff

$$(E * I)\,;\, \Xi f \;\subseteq\; E * \top\,. \tag{4.21}$$

Ξf states that dynamic frame f is preserved while the test $E * I$ ensures an initial state σ that contains at least the required locations of E. Now refining the left-hand side to $E * \top$ means these locations will not be modified in a final state σ' since E is a test.

Altogether we can now prove a central theorem of the dynamic frames theory, stating that a dynamic frame will preserve its values while modifications on a disjoint frame are performed.

Lemma 4.5.5 (Value preservation) *Assume dynamic frames f, g. If g frames test E then*

$$(E * I)\,;\,(f * g * I)\,;\,\Delta f \;\subseteq\; E * \top\,.$$

Proof. By Lemma 4.5.4, isotony, Theorem 4.5.3 and g frames E (Equation (4.21)),

$$(E * I)\,;\,(f * g * I)\,;\,\Delta f \;\subseteq\; (E * I)\,;\,(g * \Delta f) \;\subseteq\; (E * I)\,;\,(g * \top) \;\subseteq\; E * \top\,.$$

□

The abstraction of dynamic frames to sets of locations and representing them relationally as precise tests imply that they already come with the so-called *self-framing* property. It is used in the program specifications of [Kas11] to maintain that initial disjointness of dynamic frames is preserved in final states. Concretely it characterises that a dynamic frame is preserved whenever the environment does not change its value.

Lemma 4.5.6 *Every dynamic frame frames itself.*

Proof. Follows directly from $f * I \subseteq I$, isotony of ; and Theorem 4.5.3. □

Basically, dynamic frames in concrete verification applications are always assumed to be self-framing. Hence, this does not impose a restriction on the theory. We continue with an auxiliary result that is required for later calculations.

Lemma 4.5.7 *For a dynamic frame f we have $\ulcorner(\Delta f) = f * I = \ulcorner(\Xi f)$.*

A proof can be found in the Appendix.

4.5.2 Locality and Frame Accumulation

The relational structure of modifications (cf. Theorem 4.5.3) reveals that they are related to the concept of *locality* [HHM+11] that is simply characterised by the equation $Q * I = Q$ for relations Q. A relationship to this characterisation within the relational calculus has been derived in Lemma 4.3.31. In the present domain, the behaviour of relations satisfying that equation can be described as follows: at most resources in the footprint[2] of a command are modified while all other resources are left unchanged. A formal definition of footprints in the setting of local actions can be found in [RG08]. We immediately conclude as a next step

Lemma 4.5.8 *Modifications Δf satisfy locality.*

Proof. By Theorem 4.5.3, associativity of $*$, and $I * I = I$ (Lemma 4.3.6):

$$\Delta f * I \;=\; ((f\,;\top) * I) * I \;=\; (f\,;\top) * (I * I) \;=\; (f\,;\top) * I = \Delta f\,.$$

□

This is closely related to the semantics of the frame rule of separation logic. For ensuring soundness of a generalised version of that inference rule a pointfree relational variant of the frame property (cf. Definition 4.3.16) has been established, i.e.,

$$(\lceil Q \times I)\,;\rhd\,;Q \;\subseteq\; (Q \times I)\,;\rhd\,.$$

Note that we only consider the total correctness version of this property for the present approach, since program abortion and non-termination is not distinguished within the dynamic frames framework. In fact, it can be shown that relations with a precise footprint satisfy the frame property as, e.g., in the case of modifications Δf.

Lemma 4.5.9 *Modifications Δf have the frame property.*

A proof can be found in the Appendix A.

For the present treatment, Lemma 4.5.9 can be applied to prove a relational version of the *frame accumulation* law of [Kas11]. It is used in the original work a theorem that is frequently used for correctness proofs of concrete programs. We first provide the logical version of that law and describe its semantics. It is originally given as an imperative specification, i.e., a Boolean expression P that is relationally evaluated on arbitrary pairs (σ, σ') where σ denotes the initial and σ' the final state of an execution of P. The sequential composition of imperative specifications P, Q is defined by $P\,;\,Q \;\Leftrightarrow_{df}\; \exists\,\sigma'' :\; P(\sigma''/\sigma') \wedge Q(\sigma''/\sigma)$ where σ''/σ denotes the substitution of σ

[2] The minimal set of resources required for a successful execution.

with σ'', i.e., all variables will be evaluated on σ'' instead of σ. Now, the accumulation law reads as follows

$$(\,\Delta f \,\wedge\, g' \,\subseteq\, f \cup \mathsf{unused}\,)\,;\Delta g \;\;\Rightarrow\;\; \Delta f\,. \tag{4.22}$$

In a pointwise relational fashion, the accumulation law is to be understood on arbitrary pairs (σ, σ') as follows:

$$(\exists\,\sigma'' :\; \sigma\;\Delta f\;\sigma'' \,\wedge\, g_{\sigma''} \,\subseteq\, f_\sigma \cup \mathsf{unused}(\sigma) \,\wedge\, \sigma''\;\Delta g\;\sigma') \;\;\Rightarrow\;\; \sigma\;\Delta f\;\sigma'$$

where σ denotes an initial state, σ' a final state and σ'' an intermediate state due to the occurrence of ;. Note that we used $g_{\sigma''}$ above since the dynamic frame g' of Equation (4.22) denotes the final value of g on the intermediate state σ'' instead of σ'. Intuitively this law describes that whenever g in the intermediate state is bounded by f and can only increase by initially unallocated resources then the overall effect is that at most locations in f are changed in the composition $\Delta f\,;\Delta g$. Or equivalently, all allocated resources that are initially disjoint from f are preserved.

For an algebraic proof of this we need a pointfree law to characterise bounds for dynamic frames within modifications, which is of course not trivial to achieve since dynamic frames are state-dependent.

Definition 4.5.10 For dynamic frames f, g we say that g *is bounded by* f iff

$$\#\,;(f\,;\top \times I)\,;\rhd\,;(g * I) \;\;\subseteq\;\; (f\,;\top\,;(g * I) \,\times I)\,;\rhd\,.$$

Although that formula looks very complicated it is not difficult to explain. We describe its meaning within the concrete separation algebra DFSA. Of course it can be interpreted in any other adequate separation algebra, too. Assume an arbitrary pair $((\sigma_f, \sigma_I), \sigma')$ of the left-hand side of the above inequation. By this the premise reads in pointwise form as

$$\exists\,\sigma_\top, \sigma_g, \tau_I :\; \sigma_f \in f \,\wedge\, \sigma_f \,\#\, \sigma_I \,\wedge\, \sigma_\top \bullet \sigma_I = \sigma_g \bullet \tau_I = \sigma' \,\wedge\, \sigma_g \in g\,.$$

Intuitively the substate σ_f represents that part of the complete state $\sigma_f \bullet \sigma_I$ that can be changed while σ_I corresponds to the untouched part in which any changes to resources are not permitted. By assuming $\exists\,\sigma' :\; \sigma' = \sigma_\top \bullet \sigma_I$ we also know $\sigma_\top \,\#\, \sigma_I$ and hence σ_I is also disjoint from any additionally allocated resources, i.e., $dom(\sigma_I)$ is disjoint from any locations of $\mathsf{unused}(\sigma_f \bullet \sigma_I)$.

Now, the right-hand side states that

$$\exists\,\sigma_{\mathsf{rem}} :\; \sigma_f \in f \,\wedge\, \sigma' = (\sigma_g \bullet \sigma_{\mathsf{rem}}) \bullet \sigma_I \,\wedge\, \sigma_g \in g\,.$$

This means by cancellativity of the underlying separation algebra that $\sigma_\top = \sigma_g \bullet \sigma_{\mathsf{rem}}$ and $\tau_I = \sigma_{\mathsf{rem}} \bullet \sigma_I$. Hence, $\sigma_g \preceq \sigma_\top$ and $\sigma_I \preceq \tau_I$. In particular, we get $\sigma_g \# \sigma_I$, i.e., σ_g is disjoint from σ_I which in turn implies that its allocated locations can only cover locations of f and initially unallocated ones in $\mathsf{unused}(\sigma_f \bullet \sigma_I)$. The above state partitions can be depicted as in Figure 4.8.

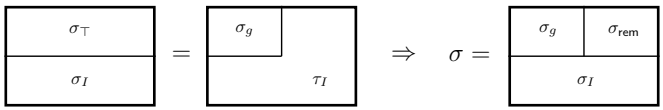


Figure 4.8: State partitions of a state σ for a bounded frame g.

Conversely, we can show using cross-split and disjointness that the underlying separation algebra satisfies the inequation of Definition 4.5.10, assuming $\sigma_g \# \sigma_I$. First, note that the premise asserts $\sigma_\top \bullet \sigma_I = \sigma_g \bullet \tau_I$ and hence $\sigma_\top \# \sigma_I$. By cross-split, i.e., Equation (4.19) we infer

$$\begin{aligned} \exists \sigma_1, \sigma_2, \sigma_3, \sigma_4 : \ & \sigma_\top = \sigma_1 \bullet \sigma_2 \ \wedge \ \sigma_I = \sigma_3 \bullet \sigma_4 \ \wedge \\ & \sigma_g = \sigma_1 \bullet \sigma_3 \ \wedge \ \tau_I = \sigma_2 \bullet \sigma_4 . \end{aligned}$$

Thus, $\sigma_g \# \sigma_I \Leftrightarrow (\sigma_1 \bullet \sigma_3 \# \sigma_3 \bullet \sigma_4) \Rightarrow \sigma_3 \# \sigma_3$ and Equation (4.20) implies that $\sigma_3 = u$. By this we immediately have $\sigma_g = \sigma_1 \wedge \sigma_I = \sigma_4$ and therefore $\sigma_\top = \sigma_g \bullet \sigma_2 \wedge \tau_I = \sigma_2 \bullet \sigma_I$. Since $\sigma_\top \# \sigma_I$ we can instantiate σ_{rem} as σ_2.

Unfortunately, Definition 4.5.10 is more complex than its logical variant which is due to implicitly expressing the particular restriction of g to unallocated resources w.r.t. f. However, with Definition 4.5.10 we now have the possibility to abstractly relate dynamic frames among each other and can continue by reasoning in an (in)equational style. By this we can summarise a central result of dynamic frames within modifications.

Theorem 4.5.11 *Assume dynamic frames where g is bounded by f. Then*

$$\Delta f \,;\, \Delta g \ \subseteq \ (f \,;\, \top \,;\, \Delta g) * I.$$

Proof. By Theorem 4.5.3, Lemma 4.3.4 and Lemma 4.5.7, g is bounded by f, $I = I;I$ and exchange $(\times/;)$, Lemma 4.5.7, Δg has the frame property and exchange $(\times/;)$ again, and definition of $*$:

$$\begin{aligned} & \Delta f \,;\, \Delta g \\ = \ & \lhd \,;\, (f \,;\, \top \times I) \,;\, \rhd \,;\, \Delta g \end{aligned}$$

$$
\begin{array}{ll}
= & \lhd\,;\#\,;(f\,;\top\,\times I)\,;\rhd\,;(g*I)\,;\Delta g \\
\subseteq & \lhd\,;(f\,;\top\,;(g*I)\,\times I)\,;\rhd\,;\Delta g \\
= & \lhd\,;(f\,;\top\,\times I)\,;((g*I)\times I)\,;\rhd\,;\Delta g \\
\subseteq & \lhd\,;(f\,;\top\,\times I)\,;(\lceil(\Delta g)\times I)\,;\rhd\,;\Delta g \\
\subseteq & \lhd\,;(f\,;\top\,;\Delta g\,\times I)\,;\rhd \\
= & (f\,;\top\,;\Delta g)*I\,.
\end{array}
$$

□

This characterises the behaviour that only the changes on the execution within f need to be considered for Δg if g is bounded by f, while all other allocated locations w.r.t a starting state will remain unchanged.

Corollary 4.5.12 (Frame Accumulation) *Assume dynamic frames f, g where g is bounded by f. Then*

$$\Delta f\,;\Delta g\,\subseteq\,\Delta f\,.$$

Proof. By Theorem 4.5.11, isotony and definition of $\top$, and Theorem 4.5.11:

$$\Delta f\,;\Delta g\,\subseteq\,(f\,;\top\,;\Delta g)*I\,\subseteq\,(f\,;\top)*I\,=\,\Delta f\,.$$

□

This result can be interpreted as a pointfree variant of the frame accumulation theorem of [Kas11] (cf. Equation (4.22)). It is applied to simplify correctness proofs of specifications by eliminating occurrences of sequential composition in combination with framing requirements.

In [Kas11] there is also the concept of *strong dynamic frames*. Such frames f come with the additional restriction on a final state σ' that $f_{\sigma'}$ can only contain locations of f_σ for a starting state σ or unallocated ones w.r.t. σ. Since the given abstractions of dynamic frames in this work imply that they are always self-framing, the modifications Δf are only able to extend f in σ' by previously unallocated locations as in [Kas11]. Hence, simple modifications Δf already coincide with the stronger variant within our abstraction.

As a final result for the abstracted theory we present the treatment also within the context of related work, i.e., local actions (cf. Section 4.3.4). In [COY07] an abstract approach to separation logic was presented that is built on separation algebras and provides a model of programs in terms of so-called *local actions*. A relationship to the relational approach and the corresponding definitions has already been provided in Section 4.3.4. In contrast to the relational calculus the local action framework works pointwise. We present in the following by the use of previous ideas about abstracting dynamic frames local action formalisations of modification and preservation framing requirements. Moreover, we show that these definitions satisfy the locality property

of local actions which is a substitute of the frame property in that approach. Finally, we give a calculational proof of the frame accumulation law within the separation algebra DFSA.

First, a state transformer definition for modifications can be obtained for a fixed set of locations f with the same ideas as in Section 4.5.1 by

$$(\Delta f)(\sigma) =_{df} \begin{cases} \Sigma * \{\sigma\big|_{\mathsf{used}-f}\} & \text{if } f \subseteq \mathsf{used}(\sigma) \\ \top & \text{otherwise}\,. \end{cases}$$

Intuitively, whenever all locations mentioned in f are allocated ones then all other used locations in σ are preserved. Otherwise, an erroneous execution is signalled by the special value $\top$ as in Section 4.3.4. Analogously, for the case of Ξf we define

$$(\Xi f)(\sigma) =_{df} \begin{cases} \Sigma * \{\sigma\big|_{f}\} & \text{if } f \subseteq \mathsf{used}(\sigma) \\ \top & \text{otherwise}\,. \end{cases}$$

According to Lemma 4.5.9, the relational version of Δf satisfies the frame property. Similar behaviour is obtained for the state transformer definition of Δf by the *locality* property of [COY07], i.e.,

$$\sigma_1 \# \sigma_2 \;\Rightarrow\; (\Delta f)\,(\sigma_1 \bullet \sigma_2) \sqsubseteq (\Delta f)\,(\sigma_1) * \{\sigma_2\}\,. \tag{4.23}$$

State transformers that satisfy Equation (4.23) are called *local actions*. The locality property has similar behaviour as the relational versions of the frame property. Due to its inclusion of $\top$ to signalise program abortion, locality is more related to the partial correctness version of the pointfree frame property. In the inequation above σ_2 represents that part of the state $\sigma_1 \bullet \sigma_2$ that will remain unchanged while σ_1 includes the footprint of Δf.

For a proof of Equation (4.23) a case distinction is needed. First assume $\sigma_1 \# \sigma_2$. If $f \not\subseteq \mathsf{used}(\sigma_1)$ then $(\Delta f)\,(\sigma_1) * \{\sigma_2\} = \top * \{\sigma_2\} = \top$ and the inequation holds. Now assume $f \subseteq \mathsf{used}(\sigma_1)$, then

$$\begin{aligned} (\Delta f)\,(\sigma_1 \bullet \sigma_2) &= \Sigma * \{\sigma_1 \bullet \sigma_2\big|_{\mathsf{used}(\sigma_1 \bullet \sigma_2)-f}\} \\ &\sqsubseteq \Sigma * \{\sigma_1\big|_{\mathsf{used}(\sigma_1)-f} \bullet \sigma_2\} \\ &= \Sigma * \{\sigma_1\big|_{\mathsf{used}(\sigma_1)-f}\} * \{\sigma_2\} \\ &= \Delta f * \{\sigma_2\}\,. \end{aligned}$$

Next we show that a treatment of the frame accumulation law is also possible using local actions. For a translation of that law into the present setting we need to define local actions that models the following restricted modification (cf. Equation (4.22)) which we provide in its logical variant by

$$\Delta(f, g) =_{df} \Delta f \wedge g' \subseteq f \cup \mathsf{unused}(\sigma)\,.$$

Note that it is evaluated on executions, i.e., pairs of states (σ, σ') where σ denotes an initial and σ' a final state, respectively. Moreover $g' = g_{\sigma'}$ generally implies the existence of a set of locations g in each state σ' of the result set $(\Delta f)(\sigma)$, if we interpret modifications as local actions. By this we restrict the local action definition of Δf as follows to get a local action for $\Delta(f, g)$:

$$(\Delta(f,g))(\sigma) =_{df} \begin{cases} \{\sigma' : \ \mathsf{used}(\sigma') = g \ \} * \Sigma * \{\sigma\big|_{\mathsf{used}-f}\} & \text{if } \ f \subseteq \mathsf{used}(\sigma) \\ \top & \text{otherwise}. \end{cases}$$

The general idea with this is to restrict the output of Δf to involve a fixed set of locations g. Another possibility would be to define a separate local action that sequentially composed with Δf restricts its output adequately. The above local action for $\Delta(f, g)$ includes the behaviour described in Definition 4.5.10 in which a bounding between dynamic frames g and f is characterised. Analogously to Δf, that state transformer is also a local action. Now, the frame accumulation law for local actions can be stated as follows:

$$\forall \sigma. \ (\Delta(f,g) \,;\Delta g)(\sigma) \ \sqsubseteq \ \Delta f(\sigma),$$

where for arbitrary local actions f, g one pointwise lifts $(f\,;g)(\sigma) =_{df} \bigsqcup\{ g(\sigma') : \sigma' \in f(\sigma)\}$ if $f(\sigma) \neq \top$ and otherwise $f\,;g$ equals $\top$. For a proof of the above inequation we assume $f \subseteq \mathsf{used}(\sigma)$ and $g \subseteq f \cup \mathsf{unused}(\sigma)$ and calculate

$$\begin{aligned}
(\Delta(f,g)\,;\Delta g)(\sigma) &= \bigsqcup\{ \Delta g(\sigma'') : \sigma'' \in \{\sigma' : \ \mathsf{used}(\sigma') = g \ \} * \Sigma * \{\sigma\big|_{\mathsf{used}-f}\}\} \\
&= \bigsqcup\{ \Delta g(\sigma' \bullet \tau \bullet \sigma\big|_{\mathsf{used}-f}) : \mathsf{used}(\sigma') = g, \tau \in \Sigma\} \\
&\sqsubseteq \bigsqcup\{ \Delta g(\sigma') * \{\tau \bullet \sigma\big|_{\mathsf{used}-f}\} : \mathsf{used}(\sigma') = g, \tau \in \Sigma\} \\
&= \bigsqcup\{ \Sigma * \{\tau \bullet \sigma\big|_{\mathsf{used}-f}\} : \tau \in \Sigma\} \\
&\sqsubseteq \bigsqcup\{ \Sigma * \{\sigma\big|_{\mathsf{used}-f}\}\} \\
&= \Sigma * \{\sigma\big|_{\mathsf{used}-f}\} \\
&= \Delta f(\sigma).
\end{aligned}$$

As further work on the presented application of dynamic frames it would be interesting to include the *overlapping conjunction* of [HV13] into this setting. Applied to assertions, this operation allows an unspecified portion of resources to be shared among two predicates. For the presented relational calculus, this would enable an abstract treatment of dynamic frames that share certain parts of their locations as e.g., in the situation when two iterators are attached to the same list as described in [Kas11]. Another possibility for this can be concrete considerations involving separation algebras that involve permissions [BCOP05]. As an example, for establishing Definition 4.5.10 with such algebras the conjecture is that the dynamic frames f, g

would have to hold full permission to each of its captured resources. Moreover, the relationships to concrete approaches [DYDG+10, PS11, JB12] and their integration into this framework have to be investigated.

Chapter 5

Transitive Separation Logic

Separation logic has been developed to allow more flexible reasoning about heap portions or, more concretely, about linked object/record structures than Hoare logic. In this chapter we give an algebraic extension of separation logic at the data structure level. We define new operations that, in addition to guaranteeing heap separation, make assumptions about the linking structure. Phenomena to be treated comprise reachability analysis, (absence of) sharing, cycle detection and preservation of substructures under destructive assignments. We demonstrate the practicality of this approach with examples of in-place list-reversal, tree rotation and threaded trees.

5.1 The Algebraic Foundation

We start with a brief example to motivate the following developments. As discussed before, the central connective of separation logic is the *separating conjunction* $p * q$ of assertions p, q. It guarantees that the addresses of the resources mentioned by p and q are disjoint. In simple settings where none of the resources of p depend on those of q and vice versa, any simple assignment to resources of p does not yield any changes of that in q. By this, one gets a compositional approach to reasoning about programs. However, the situation becomes more complex when dependencies between the set of resources exist. For a concrete example consider Figure 5.1. Clearly, from the variables x and y two singly linked lists can be accessed. Now, let p mention the starting addresses of the list records with contents $1, \ldots, 5$ and q those of the records

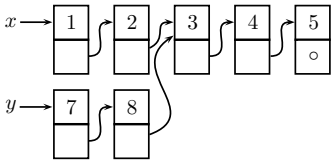


Figure 5.1: Sharing within two singly linked lists.

with contents $7, 8$. Note that $p*q$ holds, since separating conjunction only guarantees that these address sets are disjoint. However, the *contents* of the consecutive memory cells contain references to records. For those addresses there is no disjointness condition ensured. Now, if we would run, e.g., an in-place list reversal algorithm on the list accessible from x, the contents of the list accessible from y would at the same time inadvertently change, since the lists show the phenomenon of *sharing*. Therefore the goal of the following developments is to define in an abstract fashion connectives stronger than separating conjunction $*$ that ensure the absence of sharing for situations as depicted above or that restrict sharing in a way that the absence of unintended changes can be ensured. By this, we hope to facilitate reachability analysis within separation logic as, e.g., needed in garbage collection algorithms, or the detection and exclusion of cycles to guarantee termination in such algorithms.

The basic algebraic structure we start from is that of a *modal Kleene algebra* [DMS06], since it allows simple proofs in a calculational style and has proved to represent a suitable abstraction for pointer structures [Ehm04]. Another advantage is that it further allows the application of first-order automated theorem provers [HS07] and moreover captures a lot of models such as relations, regular languages or finite traces. We will introduce its constituents in several steps.

The basic foundation is given by an *idempotent semiring* denoted by $(S, +, \cdot, 0, 1)$, where $(S, +, 0)$ forms an idempotent commutative monoid and $(S, \cdot, 1)$ a monoid. We assume in the following that $\cdot$ binds tighter than $+$. Note that quantales form a special case of semirings (cf. Definition 3.1.1). We denote elements of S by $a, b, c, \ldots$. An intuitive example of an idempotent semiring is provided by the set of binary relations over some carrier set X. In that case, $+$ corresponds to relational union, $\cdot$ to relation composition, 0 to the empty relation and 1 to the identity relation I. Clearly, $+$ induces the *natural order* given by $a \leq b \Leftrightarrow_{df} a + b = b$ that relationally corresponds to the inclusion order $\subseteq$. In particular, we assume the existence of a greatest element that we denote by $\top$. It is given concretely by the universal relation. In a concrete application we can interpret elements of X as nodes of a linked data structure, such as records or objects in a list. By this, subsets of the identity relation provide a

uniform and adequate algebraic representation for sets of nodes of X. In general semirings, this approach is mimicked by sub-identity elements $p \leq 1$, called *tests* (cf. Definition 3.2.13). We recapitulate that each of these elements is requested to have a complement relative to 1, i.e., an element $\neg p$ that satisfies $p + \neg p = 1$ and $p \cdot \neg p = 0 = \neg p \cdot p$.

With tests, the abstract product $p \cdot a$ can be used to restrict an element a to links that start in nodes of p while, symmetrically, $a \cdot p$ restricts a to links ending in nodes of p. Following [DMS06], these products can be used to axiomatise algebraic variants of domain and codomain operators denoted by $\lceil\_$ and $\_\rceil$, respectively. Abstractly, for an arbitrary element a and test p they are given by the axioms

$$\begin{array}{lll} a \leq \lceil a \cdot a\,, & \lceil(p \cdot a) \leq p\,, & \lceil(a \cdot b) = \lceil(a \cdot \lceil b)\,, \\ a \leq a \cdot a\rceil\,, & (a \cdot p)\rceil \leq p\,, & (a \cdot b)\rceil = (a\rceil \cdot b)\rceil\,. \end{array}$$

These imply fundamental properties such as additivity and isotony, among others (cf. e.g [DMS06]). Note that we used the same symbol as in the concrete relational case of (reldom) in Section 4.2 since the abstract operations in fact characterise in the relational case sub-identities which are in one-to-one correspondence with the usual domain and codomain. In particular, it can be shown that the first two axioms of domain are equivalent to an abstract form of (reldom).

Built on these notions we also recapitulate the backward *diamond* operation (cf. Section 4.2) in an more abstract form. It plays a central role for our reachability analyses and is defined by $\langle a|p =_{df} (p \cdot a)\rceil$. Since this is an abstract version of the diamond operator from modal logic, an idempotent semiring with it is called *modal*. Concretely, the backward diamond $\langle a|p$ calculates all immediate successor nodes under a, starting from the set of nodes p, i.e., all nodes that are reachable within one a-step, aka the *image* of p under a. This operation distributes through union and is strict and isotone in both arguments.

Finally, to calculate reachability via arbitrarily many links or a-steps, we extend the algebraic structure to a modal *Kleene algebra* [Koz94] by an iteration operator *. It can be axiomatised by the following unfold and induction laws:

$$\begin{array}{lll} 1 + x \cdot x^* \leq x^*\,, & x \cdot y + z \leq y \;\Rightarrow\; x^* \cdot z \leq y\,, \\ 1 + x^* \cdot x \leq x^*\,, & y \cdot x + z \leq y \;\Rightarrow\; z \cdot x^* \leq y\,. \end{array}$$

This implies that a^* is the least fixed-point μ_f of $f(x) = 1 + a \cdot x$. By this we define the reachability function as follows:

$$\mathit{reach}(p, a) =_{df} \langle a^*|p\,.$$

Among other properties, *reach* distributes through $+$ in its first argument and is isotone in both arguments. Moreover we have $p \leq \mathit{reach}(p, a)$ and the *induction rule*

$p \leq q \wedge \langle a|q \leq q \;\Rightarrow\; reach(p,a) \leq q$. Further properties within a fuzzy relation algebra approach can be found in [Ehm03].

For the present chapter we basically follow the approach of [DM12b, DM13]. The last ingredient that we introduce for an adequate treatment of pointer structures is a special element within the algebra that represents the improper reference `nil` or `null`. Relationally, we can express it as the singleton relation $\Box =_{df} \{(\lightning, \lightning)\}$, where $\lightning$ is a distinguished element of the set of nodes. Such singleton sub-identity relations can abstractly be defined as *atomic* tests p.

Definition 5.1.1
A test p is called *atomic* iff $p \neq 0$ and $q \leq p \Rightarrow q = 0 \vee q = p$ for arbitrary tests q. In particular, we assume $\Box$ to be an atomic test.

Using $\Box$ we also characterise the subset of elements that have no links emanating from the pseudo-reference $\Box$ to any other address $\neq \Box$. This is a natural requirement, since the general purpose of $\Box$ is to denote a terminator reference. We refer to this property as *properness*.

Definition 5.1.2
An element a is called *proper* iff $\Box \cdot a \leq \Box$.

Note that by properness $\Box \cdot a$ is a test. We summarise some more consequences of this definition.

Lemma 5.1.3 *a_1, a_2 are proper iff $a_1 + a_2$ is proper.*

Proof. Follows immediately from distributivity and the suprema split in (3.2). □

Lemma 5.1.4 *For an element a with $\Box \cdot \lceil a = 0$ the following properties hold:*

1. a is proper, *2. $\Box \cdot a = 0$,* *3. $a = \neg\Box \cdot a$.*

Proof. For *1* and *2* we calculate $\Box \cdot a = \Box \cdot \lceil a \cdot a = 0 \cdot a = 0 \leq \Box$ by domain axioms and assumption. Finally, for *3* we have $a = 1 \cdot a = (\Box + \neg\Box) \cdot a = \Box \cdot a + \neg\Box \cdot a = \neg\Box \cdot a$ by neutrality, $\Box$ being a test, distributivity and *2*. □

Lemma 5.1.5 *If a is proper then $reach(\Box, a) = \Box$.*

Proof. First, we always have $\Box \leq reach(\Box, a)$. The other inequation $reach(\Box, a) \leq \Box$ is implied by $\langle a|\Box \leq \Box$ using the *reach* induction rule. This is shown as follows: $(\Box \cdot a)^{\urcorner} \leq \Box^{\urcorner} = \Box$ by assumption and $\Box$ being a test. □

5.2 A Stronger Notion of Separation

Following the example given in Section 5.1, we now continue with the given basics to define an adequate operation that excludes sharing within pointer structures. As a motivation, we start by another simple sharing pattern in data structures that cannot be excluded from the only use of separating conjunction $*$ as can be seen in the Figure 5.2.

Figure 5.2: Examples of sharing patterns for addresses x_1, x_2, x_3 .

Note that h_1 and h_2 satisfy the disjointness or combinability property, since $\lceil h_1 \cap \lceil h_2 = \emptyset$. But still $h = h_1 \cup h_2$ does not appear very separated from the viewpoint of reachable cells, since in the left example of Figure 5.2 both subheaps refer to the same address x_3 and in the right they form a simple cycle. This can be an undesired behaviour, since acyclicity of the data structure is a main correctness property required for many algorithms, e.g., such on linked lists or tree structures. Hence, in many cases the domain disjointness condition expressed by $\lceil h_1 \cap \lceil h_2 = \emptyset$ is too weak. Therefore we want to find, based on the given algebraic approach a stronger disjointness condition that takes such phenomena into account.

First, to simplify the description, for our new disjointness condition, we abstract from non-pointer attributes of objects, since they do not play a role for reachability questions. One can always view the non-pointer attributes of an object as combined with its address into a "super-address". Therefore we give all definitions in the following only on the relevant part of a state that affects the reachability observations.

With this abstraction, a linked object structure can be represented by an *access relation* between object addresses which we call *nodes* in the sequel. Again, we pass to the more abstract algebraic view by using elements from a modal Kleene algebra to stand for concrete access relations; hence we call them *access elements*. In the following we will denote access elements by $a, b, \ldots$. In this view, nodes are represented by atomic tests. Extending preliminary work [Ehm04, Möl99a] we give a stronger separation relation $\circledast$ on access elements.

Definition 5.2.1 (Strong disjointness)
For access elements a_1, a_2, we define the *strong disjointness relation* $\circledast$ by setting, $a = a_1 + a_2$,

$$a_1 \circledast a_2 \Leftrightarrow_{df} \; reach(\lceil a_1, a) \cdot reach(\lceil a_2, a) \leq \Box \, .$$

Intuitively, a is strongly separated into a_1 and a_2 if each address except $\Box$ that is reachable from a_1 is unreachable from a_2 w.r.t. a, and vice versa. However, since $\Box$ or, more concretely nil, is frequently used as a terminator reference in data structures, it should still be allowed to be reachable. Note that, since the result of *reach* is always a test, $\cdot$ coincides with the meet, i.e., intersection in the concrete algebra of relations. Note that the condition of strong disjointness rules out the sharing patterns of Figure 5.2. We summarise some immediate consequences.

Lemma 5.2.2 *$\circledS$ is symmetric. Moreover, $0 \circledS a$ and if a is proper then $\Box \circledS a$.*

Proof. The first claim follows from $\cdot$ coinciding with meet on tests. The rest follows from strictness of *reach* in its first argument and $p^* = 1$ for any test p and definition of *reach*. □

Since by definition we have for all tests p and access elements b that $p \leq \mathit{reach}(p, b)$, the new separation condition indeed implies the analogue of the old one, i.e., both parts are disjoint: $a_1 \circledS a_2 \Rightarrow \ulcorner a_1 \cdot \ulcorner a_2 = 0$. Finally, we can conclude

Lemma 5.2.3 *$\circledS$ is downward closed by isotony of reach, i.e., $a_1 \circledS a_2 \wedge b_1 \leq a_1 \wedge b_2 \leq a_2 \Rightarrow b_1 \circledS b_2$.*

It turns out that $\circledS$ can be characterised in a much simpler way without the implicit use of the Kleene iteration operator *. To formulate it, we define $p - q =_{df} p \cdot \neg q$ and give an auxiliary notion.

Definition 5.2.4
The *nodes* $\urcorner a \urcorner$ of an access element a are given by $\urcorner a \urcorner =_{df} \ulcorner a + a \urcorner$. A node in $a \urcorner - \ulcorner a$ is called *terminal* in a, since it has no link to any other nodes.

From the definitions it is clear that $\urcorner a + b \urcorner = \urcorner a \urcorner + \urcorner b \urcorner$ and in particular $\urcorner 0 \urcorner = 0$ and $\urcorner \Box \urcorner = \Box$. We show two further properties that link the nodes operator with reachability.

Lemma 5.2.5 *For an access element a we have*

(a) $\urcorner a \urcorner \leq \mathit{reach}(\ulcorner a, a)$,

(b) $\langle b | \urcorner a \urcorner \leq \urcorner a \urcorner \Rightarrow \mathit{reach}(\ulcorner a, a + b) = \urcorner a \urcorner$ *and hence* $\urcorner a \urcorner = \mathit{reach}(\ulcorner a, a)$.

The proof can be found in the Appendix. Trivially, the first law states that all nodes in the domain and range of an access element a are reachable from $\ulcorner a$, while the second law denotes a locality condition: If the b successors of all nodes of a are again at most nodes of a then b does not affect reachability via a. Using these theorems we can give a simpler equivalent characterisation of $\circledS$.

Lemma 5.2.6 *If a, b are proper then* $a \mathbin{⊛} b \;\Leftrightarrow\; \overline{a} \cdot \overline{b} \leq \Box$.

Proof. ($\Rightarrow$): From Lemma 5.2.5.a and isotony of *reach* we infer $\overline{a} \leq \mathit{reach}(\lceil a, a) \leq \mathit{reach}(\lceil a, a+b)$. Likewise, $\overline{b} \leq \mathit{reach}(\lceil b, a+b)$. Now the claim is immediate.
($\Leftarrow$): $\overline{a} \cdot \overline{b} \leq \Box$ implies $\overline{a} \cdot \lceil b \leq \Box$. Hence, $\langle b|\, \overline{a} = (\overline{a} \cdot b)^{\urcorner} = (\overline{a} \cdot \overline{a} \cdot \lceil b \cdot b)^{\urcorner} \leq (\overline{a} \cdot \Box \cdot b)^{\urcorner} \leq (\overline{a} \cdot \Box)^{\urcorner} \leq \overline{a}$, since b is proper and $\overline{a}, \Box$ are tests. Symmetrically $\langle a|\, \overline{b} \leq \overline{b}$ holds. Now, Lemma 5.2.5(b) tells us $\mathit{reach}(\lceil a, a+b) \cdot \mathit{reach}(\lceil b, a+b) = \overline{a} \cdot \overline{b}$, from which the claim is again immediate. □

The use of the condition in Lemma 5.2.6 instead of that in Definition 5.2.1 will considerably simplify the proofs to follow, since the Kleene * induction and unfold laws are no longer needed. Moreover, we can stay within the setting of a modal idempotent semiring using the operator $\overline{\ }$. The assumption of proper access elements is not severe, since properness is a fundamental property of pointer structures.

Lemma 5.2.7 *On proper access elements the relation $\mathbin{⊛}$ is bilinear, i.e., satisfies*

$$(a+b) \mathbin{⊛} c \;\Leftrightarrow\; a \mathbin{⊛} c \wedge b \mathbin{⊛} c \quad \textit{and} \quad a \mathbin{⊛} (b+c) \;\Leftrightarrow\; a \mathbin{⊛} b \wedge a \mathbin{⊛} c\,.$$

Proof. We use the characterisation of $\mathbin{⊛}$ from Lemma 5.2.6. First, we calculate $(a+b) \mathbin{⊛} c \;\Leftrightarrow\; \overline{a+b} \cdot \overline{c} \leq \Box \;\Leftrightarrow\; (\overline{a} + \overline{b}) \cdot \overline{c} \leq \Box \;\Leftrightarrow\; \overline{a} \cdot \overline{c} \leq \Box \wedge \overline{b} \cdot \overline{c} \leq \Box \;\Leftrightarrow\; a \mathbin{⊛} c \wedge b \mathbin{⊛} c$. The other equivalence follows from commutativity of $\mathbin{⊛}$. □

This result implies several standard laws that are crucial for calculations with the denotations for predicates or assertions, i.e., sets of states or access elements. In particular, it enables a characterisation of the interplay between the new strong separation operation and the usual separating conjunction. Similar as done in Section 3.1 for standard separation logic, the strong separation relation can be lifted to predicates.

Definition 5.2.8
For predicates P_1 and P_2, we define the *separating conjunction* $*$ and the *strongly separating conjunction* $\circledast$ by

$$\begin{aligned} P_1 * P_2 &=_{df} \{a+b : a \in P_1\,, b \in P_2\,, \lceil a \cdot \lceil b \leq 0\}\,, \\ P_1 \circledast P_2 &=_{df} \{a+b : a \in P_1\,, b \in P_2\,, a \mathbin{⊛} b\}\,. \end{aligned}$$

Moreover, we call a predicate *proper* if all its elements are proper.

Lemma 5.2.9 *The operator $\circledast$ is commutative and associative. Moreover, $P \circledast \mathsf{emp} = P$ where* $\mathsf{emp} =_{df} \{0\}$.

Proof. Commutativity is immediate from the definition. Neutrality of emp follows from $0 \mathbin{\#} a$ and by neutrality of 0 w.r.t. $+$.

For associativity, assume $a \in (P_1 \circledast P_2) \circledast P_3$, say $a = a_{12} + a_3$ with $a_{12} \mathbin{\#} a_3$ and $a_{12} \in P_1 \circledast P_2$ and $a_3 \in P_3$. Then there are a_1, a_2 with $a_1 \mathbin{\#} a_2$ and $a_{12} = a_1 + a_2$ and $a_i \in P_i$. By Lemma 5.2.7 $a_{12} \mathbin{\#} a_3$ is equivalent to $a_1 \mathbin{\#} a_3 \wedge a_2 \mathbin{\#} a_3$. Using Lemma 5.2.7 again $a_1 \mathbin{\#} a_2 \wedge a_1 \mathbin{\#} a_3 \Leftrightarrow a_1 \mathbin{\#} a_{23}$ where $a_{23} = a_2 + a_3$. Therefore $a \in P_1 \circledast (P_2 \circledast P_3)$. Hence $(P_1 \circledast P_2) \circledast P_3 = P_1 \circledast (P_2 \circledast P_3)$. □

The defined connectives are structurally similar to operations given in [HMSW09b]. Although the concrete application for that work is based on concurrency reasoning in the setting of concurrent Kleene algebras, the results still can be interpreted for our applications on pointer structures due to their abstractness. We present some of their properties and use them to characterise the interplay between separating conjunction and our stronger connective.

Lemma 5.2.10 (Exchange Laws [HMSW09b]) *Assume a semigroup $(A, +)$ and define for $P_i \subseteq A$ the predicate $P \mathbin{\circledR} Q =_{df} \{a + b : a \in P, b \in Q, a \mathrel{R} b\}$. Then for bilinear relations R and S with $R \subseteq S$ we have*

$$\begin{aligned} P_1 \mathbin{\circledR} P_2 &\subseteq P_1 \mathbin{\circledS} P_2, \\ (P_1 \mathbin{\circledS} P_2) \mathbin{\circledR} P_3 &\subseteq P_1 \mathbin{\circledS} (P_2 \mathbin{\circledR} P_3), \\ (P_1 \mathbin{\circledS} P_2) \mathbin{\circledR} (P_3 \mathbin{\circledS} P_4) &\subseteq (P_1 \mathbin{\circledR} P_3) \mathbin{\circledS} (P_2 \mathbin{\circledR} P_4). \end{aligned}$$

Since $\#$ and the standard domain disjointness condition are bilinear and $a_1 \mathbin{\#} a_2 \Rightarrow \ulcorner a_1 \cdot \ulcorner a_2 = 0$ as mentioned above, this immediately yields:

Corollary 5.2.11 *For proper predicates P_i the following inequations hold:*

$$\begin{aligned} P_1 \circledast P_2 &\subseteq P_1 * P_2, \\ (P_1 * P_2) \circledast P_3 &\subseteq P_1 * (P_2 \circledast P_3), \\ P_1 \circledast (P_2 * P_3) &\subseteq (P_1 \circledast P_2) * P_3, \\ (P_1 * P_2) \circledast (P_3 * P_4) &\subseteq (P_1 \circledast P_3) * (P_2 \circledast P_4). \end{aligned}$$

This provides useful laws for the interplay of strong separation and the standard separating conjunction. We conclude this section by some investigations on the question that arose during our developments: *why does classical separation logic get along with the weaker notion of separation rather than the stronger one?*

We will see that some aspects of our stronger notion of separation are in separation logic implicitly welded into the recursively defined data structure predicates. For an explanation of this, we concentrate on singly linked lists. They are defined according to [Rey09] by the predicate $\mathit{list}(x)$ (cf. Example 2.1.1) that states that the heap under

consideration consists of the cells of a singly linked list with starting address x. Its validity in a heap h is defined by the following clauses:

$$h \models \mathit{list}(\mathsf{nil}) \;\Leftrightarrow_{df}\; h = \emptyset\,,$$
$$x \neq \mathsf{nil} \;\Rightarrow\; (h \models \mathit{list}(x) \;\Leftrightarrow_{df}\; \exists\, y : h \models [x \mapsto y] * \mathit{list}(y))\,.$$

For simplicity, we omit the store component that records as in the definition of Example 2.1.1 the values of the program variables. Hence h has to be an empty heap when $x = \mathsf{nil}$, and a heap with at least one cell at its beginning when $x \neq \mathsf{nil}$, namely $[x \mapsto y]$.

First, note that using $\circledast$ instead of $*$ in the definition above would not work, because the heaps used are obviously not strongly separate, since their cells are connected by forward pointers to their successor cells. In the next section we introduce an approach to present such a connection within our algebra. For a more concrete description of the relationship of strong separation and the usual separation condition we define the concept of *closedness*.

Definition 5.2.12
An access element a is called *closed* iff $a^{\urcorner} \leq {}^{\ulcorner}a + \Box$.

In a closed element a there exist no dangling references. As an example, the above lists within heaps are closed as they are terminated by the value nil which abstractly corresponds to the element $\Box$. We summarise a few consequences of Definition 5.2.12.

Lemma 5.2.13 *If a_1 and a_2 are closed then $a_1 + a_2$ is also closed.*

Proof. Immediate from distributivity of domain and codomain. □

Lemma 5.2.14 *An access element a is closed iff $a^{\urcorner} - {}^{\ulcorner}a \leq \Box$.*

Proof. As tests form a Boolean subalgebra we conclude $a^{\urcorner} - {}^{\ulcorner}a \leq \Box \;\Leftrightarrow\; a^{\urcorner} \cdot \neg {}^{\ulcorner}a \leq \Box \;\Leftrightarrow\; a^{\urcorner} \leq {}^{\ulcorner}a + \Box$. □

Lemma 5.2.15 *For proper and closed a_1, a_2 with ${}^{\ulcorner}a_1 \cdot {}^{\ulcorner}a_2 = 0$ we have $a_1 \mathbin{⊛} a_2$.*

Proof. By distributivity and order theory we know

$$\overline{a_1^{\urcorner}} \cdot \overline{a_2^{\urcorner}} \leq \Box \;\Leftrightarrow\; {}^{\ulcorner}a_1 \cdot {}^{\ulcorner}a_2 \leq \Box \wedge {}^{\ulcorner}a_1 \cdot a_2^{\urcorner} \leq \Box \wedge a_1^{\urcorner} \cdot {}^{\ulcorner}a_2 \leq \Box \wedge a_1^{\urcorner} \cdot a_2^{\urcorner} \leq \Box\,.$$

The first conjunct holds by the assumption and isotony. Note that properness implies $\Box \cdot {}^{\ulcorner}a_i \leq \Box$. Hence for the second and analogously for the third conjunct we calculate

$\ulcorner a_1 \cdot a_2 \urcorner \leq \ulcorner a_1 \cdot (\ulcorner a_2 + \Box) = \ulcorner a_1 \cdot \ulcorner a_2 + \ulcorner a_1 \cdot \Box \leq \Box$. The last one reduces by distributivity and the assumptions to $\Box \cdot \Box \leq \Box$ which is trivial, since $\Box$ is a test. $\Box$

Domain-disjointness of access elements is ensured by the standard separating conjunction. It can be shown, by induction on the structure of the *list* predicate, that all access elements characterised by its analogue are closed, so that the lemma applies. This is why for a large part of separation logic the standard disjointness property suffices.

5.3 An Algebra of Linked Structures

According to [Sim06], generally recursive predicate definitions, such as the list predicate, are semantically not well defined in the classical form of separation logic. Formally, their definitions require the inclusion of fixpoint operators and additional syntactic sugar. This often makes the used assertions more complicated as e.g., by expressing reachability via anonymous addresses stored in existentially quantified variables, formulas often become very complex. To overcome this deficiency we provide operators and predicates that implicitly include such additional information, i.e., necessary correctness properties like the exclusion of sharing and reachability.

In what follows we extend our algebra following precursor work in [Ehm04, Ehm03, Möl99a, Möl92] and give some definitions to describe the shape of linked object structures, in particular of tree-like ones. We start by a characterisation of acyclicity.

Definition 5.3.1
An access element a is called *acyclic* iff for all atomic tests $p \neq \Box$ we have $p \cdot \langle a^+ | p = 0$, where $a^+ =_{df} a \cdot a^*$.

For a concrete example of this definition, one can think of an access relation a where each entry (x, y) in a^+ denotes the existence of a path from address x to y within a . Atomicity is needed to represent a single node; the definition would not work for arbitrary sets of nodes. The element $\Box$ is excluded, since it is used as a terminator reference and no structural properties are needed for it. A simpler characterisation can be given as follows.

Lemma 5.3.2 *a is acyclic iff for all atomic tests $p \neq \Box$ we have $p \cdot a^+ \cdot p = 0$.*

Proof. $p \cdot \langle a^+ | p = 0 \Leftrightarrow (p \cdot a^+)\urcorner \cdot p = 0 \Leftrightarrow (p \cdot a^+ \cdot p)\urcorner = 0 \Leftrightarrow p \cdot a^+ \cdot p = 0$ since codomain is strict. $\Box$

Next, since certain access operations are deterministic, we need an algebraic characterisation of determinacy. We borrow it from [DM01a]:

Definition 5.3.3
An access element a is *deterministic* iff $\forall p : \langle a||a\rangle p \leq p$, where the dual diamond is abstractly defined by $|a\rangle p =_{df} \ulcorner(a \cdot p)$.

A further relational characterisation of determinacy of an access relation a is given by $a^\smile \cdot a \leq 1$, where $^\smile$ is the converse operator. Since in our basic algebraic structure of semirings no general converse operation is available, we have to express the respective properties in another way. We have chosen to use the well established notion of modal operators. This way our algebra works also for other structures than relations. The roles of the expressions $a^\smile$ and a are now played by $\langle a|$ and $|a\rangle$, respectively.

Lemma 5.3.4 *If a is deterministic and $\ulcorner a$ is an atom then also $a\urcorner$ is an atom.*

A proof can be found in Appendix A. Interestingly, that proof does not presuppose that the set of all tests is an atomic lattice. Now we define our model of linked object structures.

Definition 5.3.5 (Linked structures)
We assume a finite set L of *selector names* and a modal Kleene algebra S.

- A *linked structure* is a family $a = (a_l)_{l \in L}$ of proper and deterministic access elements $a_l \in S$. This reflects that access along each particular selector is deterministic. The overall access element associated with a is then $\Sigma_{l \in L}\, a_l$, by slight abuse of notation again denoted by a; the context will disambiguate. The set of all linked structures over L is denoted by S_L. Since $\Box$ is proper and deterministic we will also view it as an element of S_L although it does not have any selectors.

- A linked structure a is a *forest* iff a is acyclic and *injective*, i.e., has maximal in-degree 1 except possibly for $\Box$. Algebraically this is expressed by the dual of the formula for determinacy, namely

  $$\forall p : |a'\rangle\langle a'|p \leq p, \quad \text{where } a' =_{df} a \cdot \neg\Box .$$

  Moreover, we define for forests a

  $$\mathit{roots}(a) =_{df} (\ulcorner a - a\urcorner) + \Box \cdot \ulcorner a .$$

  By properness and since $\Box$ is atomic, the term $\Box \cdot \ulcorner a$ equals $\Box$ when $\Box \leq a$ and is 0 otherwise.

- A forest a is called a *tree* iff $r =_{df} \mathit{roots}(a)$ is atomic and $\overline{\lceil a \rceil} = \langle a^* | r$. In this case r is called the *root* of the tree and denoted by $\mathit{root}(a)$. If additionally $L = \{\mathsf{left}, \mathsf{right}\}$ then a is a binary tree while singly linked lists arise as the special case where we have only one selector, for instance next. In this case we call a tree a *chain*. Finally, a tree a is called a *cell* if $\lceil a$ is an atomic test.

Note that $\Box$ is a tree, while 0 is not, since it has no root. But at least, 0 is a forest. For a tree a we define $\mathit{root}(a)$, derived from the above definition on forests, by

$$\mathit{root}(a) =_{df} \begin{cases} \Box & \text{if } a = \Box \\ \lceil a - a^{\smile} & \text{otherwise}. \end{cases}$$

5.4 Structural Properties of Linked Structures

As a further step we now define another separation relation that permits restricted sharing within linked structures. More precisely, we start with tree-like structures, e.g. a_1, a_2 and define them to be connected iff the root of a_2 equals one of the leafs of a_1. A main tool for expressing separateness and decomposability in such a fashion is the following.

Definition 5.4.1 (Tree combination)
Consider a selector set L. For trees $a_1, a_2 \in S_L$ we define *directed combinability* by

$$a_1 \triangleright a_2 \quad \Leftrightarrow_{df} \quad \lceil a_1 \cdot \overline{\lceil a_2 \rceil} = 0 \wedge a_1^{\smile} \cdot a_2^{\smile} \leq \Box \wedge a_1^{\smile} \cdot \lceil a_2 = \mathit{root}(a_2).$$

This relation guarantees domain disjointness and excludes occurrences of cycles, since $\lceil a_1 \cdot \overline{\lceil a_2 \rceil} = 0 \Leftrightarrow \lceil a_1 \cdot \lceil a_2 = 0 \wedge \lceil a_1 \cdot a_2^{\smile} \rceil = 0$. Hence, there can be no link from a_2 to a_1. Moreover, it excludes links from non-terminal nodes of a_1 to non-root nodes of a_2. Since a_1, a_2 are trees, it ensures that a_1 and a_2 can be combined by identifying some non-nil terminal node of a_1 with the root of a_2 (cf. Figure 5.3, where the arrows with strokes indicate in which directions links are ruled out by the definition). Note that the root cannot occur more than once in a_1.

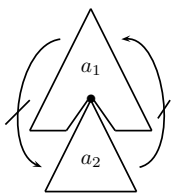


Figure 5.3: Illustration of $\triangleright$ on trees.

In particular, by Lemma 5.2.15 the second conjunct above can be dropped when both arguments are singly linked lists. We summarise some useful consequences of Definition 5.4.1.

Lemma 5.4.2 *If a is a tree then $\Box \triangleright a \Leftrightarrow \mathsf{FALSE}$ and $a \triangleright \Box \Leftrightarrow \Box \leq \overline{\lceil a \rceil}$.*

Proof. First, we have $\square \rhd a \Leftrightarrow \square \cdot \overline{\lceil a} = 0 \wedge \square \cdot \overline{a^\rceil} \leq \square \wedge \square \cdot \lceil a = root(a)$. Now, $\square \cdot \lceil a = root(a)$ implies $root(a) \leq \square$ by isotony and, since $root(a)$ is atomic and hence $\neq 0$, it must equal $\square$. By definition also $a = \square$ which immediately contradicts $\square \cdot \overline{\lceil a} = 0$.

Second, $a \rhd \square \Leftrightarrow \lceil a \cdot \square = 0 \wedge \overline{a^\rceil} \cdot \square \leq \square \wedge \overline{a^\rceil} \cdot \square = \square$. By the first result and since a is a tree the first conjunct follows from properness, the second is obvious and the third is equivalent to $\square \leq \overline{a^\rceil}$. □

Lemma 5.4.3 *Suppose trees a_1, a_2 with $a_1 \rhd a_2$. Then $root(a_1 + a_2) = root(a_1)$.*

Proof. First observe that $a_1 \neq \square$ by Lemma 5.4.2 and $a_1 \neq 0$ by definition. This implies $a_1 + a_2 \neq \square$, and we calculate

$$root(a_1 + a_2) = \lceil a_1 \cdot \neg \overline{a_1^\rceil} \cdot \neg \overline{a_2^\rceil} + \lceil a_2 \cdot \neg \overline{a_1^\rceil} \cdot \neg \overline{a_2^\rceil}.$$

The first summand reduces to $\lceil a_1 \cdot \neg \overline{a_1^\rceil} = root(a_1)$, since $a_1 \rhd a_2$ implies $\lceil a_1 \cdot \overline{a_2^\rceil} = 0$, i.e., $\lceil a_1 \leq \neg \overline{a_2^\rceil}$. The second summand is, by definition, equal to $root(a_2) \cdot \neg \overline{a_1^\rceil}$. Since $a_1 \rhd a_2$ implies $root(a_2) \leq \overline{a_1^\rceil}$, this summand reduces to 0. □

Summarised, for combinable trees a_1, a_2 the root of $a_1 + a_2$ is determined by a_1. Note that we have not assumed that $a_1 + a_2$ forms a tree although *root* is only defined for trees. We used *root* here instead of its concrete definition for better readability.

Since the directed disjointness relation $\rhd$ is defined only on tree-like structures, we extend it now to arbitrary forests. For this we assume in the following that any forest a can be represented by a finite summation of trees a_i, i.e., $a = \sum a_i$.

Definition 5.4.4 (Forest combination)
Consider a selector set L and let $a, b \in S_L$ be forests with $a = \sum a_i$ and $b = \sum b_j$, where the a_i and b_j are the constituent trees with $a_{i_1} \circledast a_{i_2}$ $(i_1 \neq i_2)$ and $b_{j_1} \circledast b_{j_2}$ $(j_1 \neq j_2)$. Then we define *directed combinability* by

$$a \rhd b \;\Leftrightarrow_{df}\; \exists\, i, j :\; a_i \rhd b_j \;\wedge\; \left(\sum_{k \neq i} a_k \right) \circledast b_j .$$

Note that $\rhd$ on the constituent trees a_i, b_j is given by Definition 5.4.1. We refer to these components by numbers $i \in \mathbb{N}$ and to a particular selector $l \in L$ of its access element by $(a_i)_l$. Definition 5.4.4 requires at least two constituent trees of the forests a and b to be connected w.r.t. $\rhd$ while all previously unconnected trees remain strongly disjoint (cf. Figure 5.4). Moreover one has to exclude the case that another tree a_m gets connected with b_j as this would yield only a directed acyclic graph and introduce prohibited sharing.

We now show that $\rhd$ guarantees preservation of linked structures under $+$.

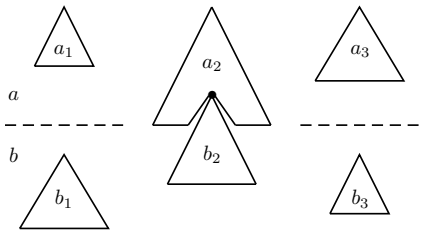


Figure 5.4: $\triangleright$- combination of forests a, b.

Lemma 5.4.5 *Let a_1, a_2 be arbitrary elements of a modal semiring.*

(a) *If the a_i are deterministic and $\lceil a_1 \cdot \lceil a_2 = 0$ then also $a_1 + a_2$ is deterministic.*

(b) *If the a_i are injective and $a_1{}^{\rceil} \cdot a_2{}^{\rceil} \leq \Box$ then also $a_1 + a_2$ is injective.*

(c) *If the a_i are acyclic and $a_2{}^{\rceil} \cdot \lceil a_1 = 0$ then also $a_1 + a_2$ is acyclic.*

Proof.

(a) By distributivity, $\langle a_1 + a_2 || a_1 + a_2 \rangle p \leq p$, since $\langle a_i || a_i \rangle p \leq p$ and $\langle a_2 || a_1 \rangle p \leq 0 \wedge \langle a_1 || a_2 \rangle p \leq 0$ by $\lceil a_1 \cdot \lceil a_2 = 0$.

(b) By definition and distributivity we have $(a_1 + a_2)' = (a_1 + a_2) \cdot \neg\Box = a_1' + a_2'$ with a' as defined in Definition 5.3.5. Now we can reason symmetrically to Part (a).

(c) Assume an arbitrary atomic test $p \neq \Box$. We show that $p \cdot (a_1 + a_2)^+ \cdot p = 0$. First note that if $a_2{}^{\rceil} \cdot \lceil a_1 = 0$ then $(a_1 + a_2)^+ = {a_1}^+ + {a_1}^+ \cdot {a_2}^+ + {a_2}^+$. This follows using $(x + y)^* = x^* \cdot (y \cdot x^*)^*$, domain properties and the definition of $\_^+$.

Hence, it remains to show $p \cdot {a_1}^+ \cdot p = 0 \wedge p \cdot {a_1}^+ \cdot {a_2}^+ \cdot p = 0 \wedge p \cdot {a_2}^+ \cdot p = 0$. The first and last conjuncts follow from the assumption. If the second conjunct would be false, then necessarily $0 \neq p \cdot {a_1}^+ = p \cdot a_1 \cdot a_1^*$ and hence $p \cdot \lceil a_1 \neq 0$. Likewise, $p \cdot a_2{}^{\rceil} \neq 0$. Since p is an atom, these two conditions are equivalent to $p \leq \lceil a_1$ and $p \leq a_2{}^{\rceil}$, respectively, and hence imply $p \leq a_2{}^{\rceil} \cdot \lceil a_1$. This is a contradiction to $a_2{}^{\rceil} \cdot \lceil a_1 = 0$ and atomicity of p.

□

Corollary 5.4.6 *Consider a selector set L. If $a_1, a_2 \in S_L$ are linked structures with $\lceil a_1 \cdot \overline{\lceil a_2}^{\rceil} = 0$ and $a_1{}^{\rceil} \cdot \lceil a_2 \leq \Box$ then also $a_1 + a_2$ is a linked structure in S_L.*

Proof. Properness of $a_1 + a_2$ follows from Lemma 5.1.3. The remaining properties required of $a_1 + a_2$ are implied by Lemma 5.4.5. □

Lemma 5.4.7 *If a_1, a_2 are trees with $a_1 \triangleright a_2$ and then $a_1 + a_2$ is again a tree whose root is that of a_1 .*

Proof. Since $a_1 \triangleright a_2$ implies the assumptions of Corollary 5.4.6, we know that $a_1 + a_2$ is a linked structure. Moreover, by Lemma 5.4.3 we have that $root(a_1 + a_2) = root(a_1)$ and thus is atomic. It remains to show $\overline{a_1 + a_2} = \langle (a_1 + a_2)^* | root(a_1)$. We know that $\overline{a_1 + a_2} = \overline{a_1} + \overline{a_2}$.

($\leq$): By the assumptions and isotony, $\overline{a_1} = \langle {a_1}^* | root(a_1) \leq \langle (a_1 + a_2)^* | root(a_1)$. Second, again by the assumptions, $\langle b | \langle a | p = \langle a \cdot b | p$ and isotony, we obtain

$$\begin{aligned} \overline{a_2} &= \langle {a_2}^* | root(a_2) \leq \langle {a_2}^* | \overline{a_1} = \langle {a_2}^* | \langle {a_1}^* | root(a_1) \\ &= \langle {a_1}^* \cdot {a_2}^* | root(a_1) \leq \langle (a_1 + a_2)^* | root(a_1) \,. \end{aligned}$$

($\geq$): For abbreviation, set $q =_{df} \overline{a_1 + a_2} = \langle {a_1}^* | root(a_1) + \langle {a_2}^* | root(a_2)$. Using diamond induction, $\langle (a_1 + a_2)^* | root(a_1) \leq q$ is implied by $root(a_1) \leq q$ and $\langle a_1 + a_2 | q \leq q$. The first conjunct is clear while the second is by distributivity and again $\langle b | \langle a | p = \langle a \cdot b | p$ equivalent to

$$\langle {a_1}^* \cdot a_1 | root(a_1) + \langle {a_1}^* \cdot a_2 | root(a_1) + \langle {a_2}^* \cdot a_1 | root(a_2) + \langle {a_2}^* \cdot a_2 | root(a_2) \leq q \,.$$

By suprema split w.r.t. $+$ the inequation for the first and last summands are clear. The remaining ones are treated by

$$\begin{aligned} \langle {a_1}^* \cdot a_2 | root(a_1) &= \langle a_2 | root(a_1) + \langle {a_1}^* \cdot a_1 \cdot a_2 | root(a_1) \\ &= \langle {a_1}^* \cdot a_1 \cdot root(a_2) \cdot a_2 | root(a_1) \\ &= \langle a_2 | ((root(a_1) \cdot {a_1}^* \cdot a_1)^{\urcorner} \cdot root(a_2)) \\ &\leq \langle a_2 | root(a_2) \end{aligned}$$

and $\langle {a_2}^* \cdot a_1 | root(a_2) = \langle a_1 | root(a_2) + \langle {a_2}^* \cdot a_2 \cdot a_1 | root(a_2) = 0$. □

Corollary 5.4.8 *Since lists are a special case of trees, Lemma 5.4.7 also holds for lists.*

Corollary 5.4.9 *If a_1, a_2 are forests and $a_1 \triangleright a_2$ or $a_1 \mathbin{⊛\!\!\!\!\!\!\#} a_2$ holds then also $a_1 + a_2$ is a forest.*

Proof. Immediate from Lemma 5.4.7 and the definition of $\triangleright$ on forests. □

As before we can again lift the relation $\triangleright$ to predicates. First, we define the following special predicates

$$\begin{aligned} [\![\mathsf{cell}]\!] &=_{df} \{a : a \text{ is a cell}\} \,, \\ [\![\mathsf{list}]\!] &=_{df} \{a : a \text{ is a chain}\} \,, \\ [\![\mathsf{tree}]\!] &=_{df} \{a : a \text{ is a tree}\} \,, \\ [\![\mathsf{forest}]\!] &=_{df} \{a : a \text{ is a forest}\} \,. \end{aligned}$$

Clearly, $[\![\,\mathsf{cell}\,]\!] \cap S_{\mathsf{next}} \subseteq [\![\,\mathsf{list}\,]\!] \subseteq [\![\,\mathsf{tree}\,]\!] \subseteq [\![\mathsf{forest}]\!]$ and $[\![\,\mathsf{cell}\,]\!] \subseteq [\![\,\mathsf{tree}\,]\!]$. Note that we used the subset $[\![\,\mathsf{cell}\,]\!] \cap S_{\mathsf{next}}$ since by definition no restriction on the set of selectors L is mentioned.

Definition 5.4.10 (Directed combination)
For a selector set L and $P, Q \subseteq \mathsf{forest} \cap S_L$ we define *directed combinability* $\circledcirc$ by

$$P \circledcirc Q \;=_{df}\; \{\, a_1 + a_2 : a_1 \in P,\, a_2 \in Q,\, a_1 \triangleright a_2 \,\}\,.$$

To avoid excessive notation, in the sequel we tacitly assume that all predicates involved in our formulas are restricted to the same set of selectors as in this definition. This definition allows, conversely, also talking about decomposability: If $a \in P_1 \circledcirc P_2$ then a can be split into two disjoint parts a_1, a_2 such that $a_1 \triangleright a_2$ holds. For better readability we omit the braces $[\![\_]\!]$ in what follows.

Lemma 5.4.11 $\mathsf{forest} \circledcirc \mathsf{forest} \subseteq \mathsf{forest}$, $\mathsf{tree} \circledcirc \mathsf{tree} \subseteq \mathsf{tree}$ *and* $\mathsf{list} \circledcirc \mathsf{list} \subseteq \mathsf{list}$. *As particular cases we have* $\mathsf{cell} \circledcirc \mathsf{list} \subseteq \mathsf{list}$, $\mathsf{tree} \circledcirc \mathsf{cell} \subseteq \mathsf{tree}$ *and* $\mathsf{cell} \circledcirc \mathsf{tree} \subseteq \mathsf{tree}$.

Lemma 5.4.12 *Let* $P, Q, R \subseteq \mathsf{tree}$ *then*

$$\begin{array}{rcl} P \circledcirc (Q \circledcirc R) & \subseteq & (P \circledcirc Q) \circledcirc R\,, \\ P \circledcirc (Q \circledcirc R) & \subseteq & (P \circledast R) * Q\,, \\ (P \circledcirc Q) \circledast R & \subseteq & P \circledcirc (Q \circledast R)\,, \\ P \circledast (Q \circledcirc R) & \subseteq & (P \circledast Q) \circledcirc R\,. \end{array}$$

Proof. We start with the first two laws. Assume $a_1 \in P$, $a_2 \in Q$, $a_3 \in R$ and $a_1 \triangleright (a_2 + a_3)$ and $a_2 \triangleright a_3$. By Lemma 5.4.2 we know $a_1, a_2 \neq \Box$. Moreover, by Lemma 5.4.7 $a_2 + a_3$ is a tree with $\mathit{root}(a_2 + a_3) = \mathit{root}(a_2)$. Now, $a_1 \triangleright (a_2 + a_3)$ implies $a_1^{\neg} \cdot \ulcorner a_2 + a_1^{\neg} \cdot \ulcorner a_3 = \ulcorner a_2 - a_2^{\neg}$. Multiplying this equation by $\ulcorner a_2$ and using that $a_2 \triangleright a_3$ implies $\ulcorner a_3 \cdot \ulcorner a_2 = 0$ we obtain $a_1^{\neg} \cdot \ulcorner a_2 = \ulcorner a_2 - a_2^{\neg} = \mathit{root}(a_2)$. Hence, $a_1^{\neg} \cdot \ulcorner a_3 = 0$, since $\mathit{root}(a_2)$ is atomic. By this we can immediately derive from distributivity and the definitions that $a_1 \triangleright a_2 \wedge (a_1 + a_2) \triangleright a_3$ and $a_1 \mathbin{\#} a_3 \wedge \ulcorner(a_1 + a_3) \cdot \ulcorner a_2 \leq 0$, which shows the first two laws.

For the third law, assume $a_1 \triangleright a_2$ and $(a_1 + a_2) \mathbin{\#} a_3$ which is equivalent to $a_1 \mathbin{\#} a_3 \wedge a_2 \mathbin{\#} a_3$. Note, that $a_2 + a_3$ is a forest. Hence by Definition 5.4.4 the claim is immediate.

Finally, the last inequation follows directly from bilinearity of $\#$ and the definition of $\triangleright$ on forests. $\Box$

5.5 Assertions and Program Commands

We now define programming constructs to treat concrete verification examples. As a first step we extend our predicates by a possibility of directly addressing the roots of the characterised structures. For this we define, similar to standard separation logic, so-called *stores*.

Definition 5.5.1
A *store* is a partial mapping from program identifiers to nodes, i.e., atomic tests. The domain of a store s is denoted by $dom(s)$. A *state* is a pair (s,a) with a store s and a linked structure a. For an identifier i and a sequence $l = l_1 \dots l_n \in L^+$ of selector names, the semantics of the expression $\mathsf{i}.l$ w.r.t. a state (s,a) is defined as

$$[\![\, \mathsf{i}.l \,]\!]_{(s,a)} =_{df} \begin{cases} \langle a_{l_1} \cdot \ldots \cdot a_{l_n} | s(\mathsf{i}) & \text{if } \mathsf{i} \in dom(s) \\ 0 & \text{otherwise.} \end{cases}$$

Note that $\langle a_{l_1} \cdot \ldots \cdot a_{l_n} | s(\mathsf{i})$ is either an atomic test or 0 by determinacy of each access element a_{l_i}. In particular, we always have $[\![\, \mathsf{i}.l \,]\!]_{(s,a)} \leq \overline{a}$.

Definition 5.5.2
For an identifier i and a predicate $P \subseteq \mathsf{tree}$ we define its extension $P(\mathsf{i})$ to states by

$$P(\mathsf{i}) =_{df} \{(s,a) : a \in P,\, \mathsf{i} \in dom(s),\, root(a) = s(\mathsf{i})\}.$$

By this we can refer to the root of an access element a in predicates about tree-like structures. If we are not interested in the root nodes we will, by slight abuse of notation, simply write P also to mean the extension of P to states, i.e., $P =_{df} \{(s,a) : a \in P\}$. In particular, for an operator $\circ \in \{=, \neq\}$ and $l, m \in L^+$, we define a denotational semantics for special predicates by

$$\begin{aligned} [\![\, \mathsf{i} \circ \square \,]\!] &=_{df} \{(s,a) : \mathsf{i} \in dom(s),\, s(\mathsf{i}) \circ \square\}, \\ [\![\, \mathsf{i}.l \circ \square \,]\!] &=_{df} \{(s,a) : 0 \neq [\![\, \mathsf{i}.l \,]\!]_{(s,a)} \circ \square\}, \\ [\![\, \mathsf{i}.l = \mathsf{j}.m \,]\!] &=_{df} \{(s,a) : 0 \neq [\![\, \mathsf{i}.l \,]\!]_{(s,a)} = [\![\, \mathsf{j}.m \,]\!]_{(s,a)}\}. \end{aligned}$$

The mechanism of predicate extension cannot be used with expressions e involving selector chains. Simply setting $P(e) =_{df} \{(s,a) : a \in P, root(a) = [\![\, e \,]\!]_{(s,a)}\}$ would, for instance, not work in a formula like $P(\mathsf{i}) \circledcirc Q(\mathsf{i}.l)$, since by the definition of $\circledcirc$ we cannot have $s(\mathsf{i}) \leq \ulcorner a$ with $a \in Q$ due to the implicit separation condition. Hence, we require a global view of the considered states and hence use $P(\mathsf{i}) \circledcirc Q(\mathsf{i}.l)$ as an abbreviation for $(P(\mathsf{i}) \circledcirc Q(\mathsf{j})) \cap [\![\, \mathsf{j} = \mathsf{i}.l \,]\!]$ where j is a fresh identifier. Note that within logical assertions $\cap$ coincides with $\wedge$ and hence enables an evaluation of expression

on the complete state. The predicate $\mathsf{j} = \mathsf{i}.l$ is used to name an otherwise anonymous node within the structure rooted in i. We remark that standard separation logic does not allow heap dependent expressions (cf. Section 2.1). Hence they can be completely evaluated on the store component. Our treatment rather follows the dynamic frames approach [Kas11] (cf. Section 4.5) that also allows heap-dependent expressions as e.g., stated in [PS12].

The extension of predicates to states with stores allows placing side conditions on the root elements of predicates in formulas. This has many useful consequences. We summarise a few association properties in the following.

Lemma 5.5.3 *Let* $\mathsf{i}, \mathsf{j}, \mathsf{k}$ *be identifiers and* $\{\square\} \not\subseteq P, Q, R \subseteq \mathsf{tree}$. *Then*

(a) $(P(\mathsf{i}) \circledcirc Q(\mathsf{j})) \circledcirc R(\mathsf{k}) = P(\mathsf{i}) \circledcirc (Q(\mathsf{j}) \circledcirc R(\mathsf{k}))$ *if* $\exists\, l \in L^+ : \mathsf{j}.l = \mathsf{k}$,

(b) $(P(\mathsf{i}) \circledcirc Q) \circledast R(\mathsf{j}) = P(\mathsf{i}) \circledcirc (Q \circledast R(\mathsf{j}))$ *if* $\forall\, l \in L^+ : \mathsf{i}.l \neq \mathsf{j}$,

(c) $(P(\mathsf{i}) \circledcirc Q(\mathsf{j})) \circledcirc R(\mathsf{k}) = P(\mathsf{i}) \circledcirc (Q(\mathsf{j}) \circledast R(\mathsf{k}))$ *if* $\mathsf{j} = \mathsf{i}.l \wedge \mathsf{k} = \mathsf{i}.m \wedge l, m \in L$,

(d) $(P(\mathsf{i}) \circledast Q(\mathsf{j})) \circledcirc R(\mathsf{k}) = P(\mathsf{i}) \circledast (Q(\mathsf{j}) \circledcirc R(\mathsf{k}))$ *if* $\exists\, l \in L^+ : \mathsf{j}.l = \mathsf{k}$,

where $l \in L^+$ *denotes that* l *is a non-empty sequences of selector names.*

The proof can be found in the Appendix. The side conditions on the variables represent reachability requirements that are needed to obtain the other inclusion of Lemma 5.4.12. As future work it would be interesting to investigate their integration into suitable operations as this would facilitate the treatment. However, this form suffices for our purposes in proofs of programs. As a next step we consider the special case of chains.

Corollary 5.5.4 *For arbitrary* $P, Q, R \subseteq \mathsf{list}$ *and identifier* i *we have*

$$(P(\mathsf{i}) \circledcirc Q(\mathsf{i.next})) \circledcirc R(\mathsf{i.next.next}) = P(\mathsf{i}) \circledcirc (Q(\mathsf{i.next}) \circledcirc R(\mathsf{i.next.next}))\,,$$

i.e., $\circledcirc$ *is associative on lists.*

Proof. This follows from Lemma 5.5.3(a) by setting $\mathsf{j} = \mathsf{i.next}$ and $\mathsf{j.next} = \mathsf{k}$. □

Next we want to give semantics to program commands, in particular, to assignments of the form $\mathsf{i}.l := e$. To this end we enrich our algebraic setting by another ingredient, namely by *twigs*, i.e., abstract representations of single edges in the graph corresponding to a linked structure. Special assignments of the above form will add or delete such twigs.

Definition 5.5.5 (Twigs)
Assume atomic tests with $p \cdot q = 0 \wedge p \cdot \Box = 0$. We define a *twig* by $p \mapsto q =_{df} p \cdot \top \cdot q$ where $\top$ denotes the greatest element of the underlying modal Kleene algebra. The corresponding *update* (cf. Equation 2.1) of a linked structure a is given by $(p \mapsto q) \mid a =_{df} (p \mapsto q) + \neg p \cdot a$. We assume that $\mid$ binds tighter than $+$ but less tight than $\cdot$.

We called the elements $p \mapsto q$ twigs as they intuitively corresponds to the least non-nil components in trees or forests. Note, that by $p, q \neq 0$ also $p \mapsto q \neq 0$. Intuitively, in $(p \mapsto q) \mid a$, the single node of p is connected to the single node in q, while a is restricted to links that start from $\neg p$ only. Assuming the *Tarski rule*, i.e., $\forall a : a \neq 0 \Rightarrow \top \cdot a \cdot \top = \top$, we can easily infer for a twig $(p \mapsto q)^{\urcorner} = q$ and $\ulcorner(p \mapsto q) = p$.

Lemma 5.5.6 $\overline{p \mapsto q} = p + q$ *and* $root(p \mapsto q) = p$.

Proof. The first result is trivial. Second, $root(p \mapsto q) = \ulcorner(p \mapsto q) \cdot \neg(p \mapsto q)^{\urcorner} = p \cdot \neg q = p$, since $p \cdot q = 0 \Leftrightarrow p \leq \neg q$ by shunting. □
Note that by $a = 0 \Leftrightarrow \ulcorner a = 0$, cells are always non-empty.

Lemma 5.5.7 *For a cell a we have $root(a) = \ulcorner a$, hence $\neg root(a) \cdot a = 0$.*

Proof. By definition $root(a) \leq \ulcorner a$ and $root(a) \neq 0$. Thus $root(a) = \ulcorner a$. □

Lemma 5.5.8 *Twigs $p \mapsto q$ are cells.*

Proof. By assumption, $\ulcorner(p \mapsto q) = p$ is atomic and $\neq \Box$, hence proper. Moreover, $reach(p, p \mapsto q) = \overline{p \mapsto q} = p + q$, acyclicity holds by $p \cdot q = 0$. To show determinacy we conclude for arbitrary tests s:

$$q \cdot s \leq q \;\Rightarrow\; q \cdot s = 0 \vee q \cdot s = q \;\Leftrightarrow\; q \cdot s = 0 \vee q \leq s\,.$$

Hence, $\langle p \mapsto q || p \mapsto q \rangle s \leq \langle p \mapsto q | p \leq q \leq s$. The calculation for injectivity is analogous. □
Now, we can summarise a few consequences that will be used in the examples to come.

Corollary 5.5.9 $[\![\mathsf{i} \neq \Box]\!] \cap \mathsf{list}\,(\mathsf{i}) = \mathsf{cell}\,(\mathsf{i}) \circledcirc \mathsf{list}$ *and* $[\![\mathsf{i} = \Box]\!] \cap \mathsf{list}\,(\mathsf{i}) = \{\Box\}$.

Proof. We only show $\mathsf{list}\,(\mathsf{i}) = \mathsf{cell}\,(\mathsf{i}) \circledcirc \mathsf{list}$, since the second result is obvious. The $\supseteq$-direction follows from Lemma 5.4.7. For $\subseteq$ we know by the assumption $\mathsf{i} \neq \Box$

and the definitions that $a \neq \Box$ for all $(s,a) \in \mathsf{list}(\mathsf{i})$. Since a is a chain and therefore acyclic, we can write $a = (root(a) \mapsto root(b)) + b$ where $b =_{df} \neg root(a) \cdot a$. Note that by Lemma 5.5.8 $(root(a) \mapsto root(b)) \in \mathsf{cell}$. By this one can show $b \in \mathsf{list}$ and $(root(a) \mapsto root(b)) \triangleright b$. $\Box$

Corollary 5.5.10 $[\![\mathsf{i.left} \neq \Box]\!] \cap [\![\mathsf{i.right} \neq \Box]\!] \cap \mathsf{tree(i)} = \mathsf{cell}(\mathsf{i}) \circledcirc (\mathsf{tree(i.left)} \circledast \mathsf{tree(i.right)})$.

Proof. A proof can be constructed similarly as for Corollary 5.5.9. $\Box$

Now, we are ready to provide definitions for the concrete meaning of program *commands*. They are modelled denotationally as in Section 4 as relations between states. For assignments of the form $\mathsf{i}.l := e$, we use twigs (cf. Definition 5.5.5) to describe updates of linked structures by adding or changing links. In particular, we use expressions e of the form $\langle var \rangle.l$ where var is an arbitrary variable and $l \in L^+$.

Definition 5.5.11 (Commands on linked structures)
In the following we assume an identifier i, a selector set L, a selector name $l \in L$ and an expression e for which $[\![e]\!]_{(s,a)}$ is always an atomic test. For a linked structure $a \in S_L$ we abbreviate the subfamily $(a_k)_{k \in L-\{l\}}$ by a_{L-l}. Then we define a relational semantics for commands on linked structures by

$$
\begin{array}{lll}
[\![\mathsf{i} := e]\!] & =_{df} & \{\, ((s,a),(s[\mathsf{i} \leftarrow p],a)) : \mathsf{i} \in dom(s),\ p = [\![e]\!]_{(s,a)} \,\}, \\
[\![\mathsf{i}.l := e]\!] & =_{df} & \{\, ((s,a),(s,(s(\mathsf{i}) \mapsto [\![e]\!]_{(s,a)})|a_l + a_{L-l}) : \mathsf{i} \in dom(s), \\
 & & \qquad s(\mathsf{i}) \neq \Box,\ s(\mathsf{i}) \leq \ulcorner a_l \,\}, \\
[\![\mathsf{i} := \mathsf{new\ cell}()]\!] & =_{df} & \{\, ((s,a),(s[\mathsf{i} \leftarrow p],(p \mapsto \Box)|a) : \mathsf{i} \in dom(s),\ p \leq \neg \ulcorner a, \\
 & & \qquad p \text{ is an atomic test},\ p \neq \Box \,\}, \\
[\![\mathsf{delete}(\mathsf{i})]\!] & =_{df} & \{\, ((s,a),(s,\neg p \cdot a)) : p = s(\mathsf{i}),\ p \leq \ulcorner a,\ \mathsf{i} \in dom(s),\ p \neq \Box \,\}.
\end{array}
$$

The definitions closely corresponds to the total correctness semantics of the separation logic commands given in Section 2.2 as we included identifier and selector assignments, single cell allocation and deletion. Similar to Section 4.1 we can obtain a partial correctness version for validating the frame rule by adding a distinguished state that denotes program abortion. This further requires an inclusion of aborting executions to the commands of Definition 5.5.11 whenever the conditions involving $\ulcorner a$ or $\ulcorner a_l$ are not satisfied.

One particular difference to standard separation logic is that we allow assignments of the form $\mathsf{i}.l := \mathsf{j}.m$ for identifiers i, j and selectors l, m, i.e., a dereferencing on both sides of the assignment operator $:=$. In separation logic this can be mimicked by the use of a temporal store variable. In general selector assignments do not preserve the tree structure. We provide sufficient conditions for that in the form of Hoare triples in the next section.

5.6 Inference Rules

As already done in Section 4.2, we follow the presented relational approach for correctness proofs of our extension to separation logic. We encode sets of access elements or predicates P as subidentity relations of the form $\{(\sigma,\sigma) : \sigma \in P\}$ where $\sigma = (s,a)$ for some store s and linked structure a. For easier readability we will now omit the $[\![\_]\!]$-brackets and do not distinguish assertions and commands from their corresponding relations notationally. We will use Lemma 4.3.26 for a relational encoding of Hoare triples, i.e., for predicates P,Q and command C we use

$$\{P\}\,C\,\{Q\} \Leftrightarrow_{df} \widetilde{P};C \subseteq C;\widetilde{Q} \Leftrightarrow \widetilde{P};C \subseteq \top;\widetilde{Q}\,.$$

For concentrating on the main details of our extension to separation logic we facilitate the partial correctness treatment by excluding separate calculations on abort free predicates $\widetilde{P}$ as in Section 4.2. With the previous basics it is not difficult to give corresponding and analogous calculations within the present setting.

5.6.1 Selector Assignments

First, we abbreviate the following rules by introducing some syntactic sugar. For expressions e,e' and operators $\circ \in \{*, \circledast, \circledcirc\}$, we replace formulas of the form $Q \circ P(e) \wedge e' = e$ by $Q \circ P(e,e')$. By this we can explicitly show expressions that are aliases for the same root node. For instance, we can abbreviate the rule

$$\begin{array}{l}\{\ P(\mathsf{j}) \circledcirc Q(\mathsf{j}.l)\ \}\\ \mathsf{i} := \mathsf{j}.l;\\ \{\ P(\mathsf{j}) \circledcirc Q(\mathsf{j}.l) \wedge \mathsf{i} = \mathsf{j}.l\ \}\end{array} \quad \text{to} \quad \begin{array}{l}\{\ P(\mathsf{j}) \circledcirc Q(\mathsf{j}.l)\ \}\\ \mathsf{i} := \mathsf{j}.l;\\ \{\ P(\mathsf{j}) \circledcirc Q(\mathsf{j}.l,\mathsf{i})\ \}\,.\end{array}$$

Lemma 5.6.1 *For predicates $P,Q,R \subseteq \mathsf{tree}$, identifiers i,j and selector $l \in L$:*

$$\begin{array}{lll}\{\ (P(\mathsf{i}) \circledcirc Q(\mathsf{i}.l)) \circledast R(\mathsf{j})\ \} & \{\ P(\mathsf{i}) \circledast R(\mathsf{j}) \wedge \mathsf{i}.l = \square\ \} & \{\ P(\mathsf{i}) \circledcirc Q(\mathsf{i}.l)\ \}\\ \mathsf{i}.l := \mathsf{j}; & \mathsf{i}.l := \mathsf{j}; & \mathsf{i}.l := \square;\\ \{\ (P(\mathsf{i}) \circledcirc R(\mathsf{j},\mathsf{i}.l)) \circledast Q\ \}, & \{\ P(\mathsf{i}) \circledcirc R(\mathsf{j},\mathsf{i}.l)\ \}, & \{\ P(\mathsf{i}) \circledast Q \wedge \mathsf{i}.l = \square\ \}\,.\end{array}$$

Proof. We only give a proof of the leftmost rule. The remaining ones can be proved similarly. Assume trees $a_1 \in P \wedge a_2 \in Q \wedge a_3 \in R$ with $a_1 \triangleright a_2 \wedge a_1 \mathbin{\#\!\!\circledcirc} a_3 \wedge a_2 \mathbin{\#\!\!\circledcirc} a_3 \wedge a = a_1 + a_2 + a_3$. We decompose each a_i into its l-part $b_i =_{df} (a_i)_l$ and the rest $c_i =_{df} (a_i)_{L-l}$ and show $((root(a_1) \mapsto root(a_3))|b_1 + c_1) \mathbin{\#\!\!\circledcirc} a_2$. This is equivalent to $c_1 \mathbin{\#\!\!\circledcirc} c_2 \wedge (root(a_1) \mapsto root(a_3)) \mathbin{\#\!\!\circledcirc} b_2 \wedge (\neg root(a_1) \cdot b_1) \mathbin{\#\!\!\circledcirc} b_2$.

By assumption we know $(root(a_1) \cdot b_1)^{\urcorner} = root(a_2)$. This implies by the injectivity property of trees and atomicity that $(\neg root(a_1) \cdot b_1)^{\urcorner} \cdot \ulcorner a_2 = 0$. Hence, together with

$a_1 \circledcirc a_2$ we have $(\neg root(a_1) \cdot b_1) \circledast b_2$. By determinacy and again the assumption on the roots, $a_1^{\urcorner} \cdot {}^{\ulcorner}a_2 = root(a_2)$ is equivalent to $b_1^{\urcorner} \cdot {}^{\ulcorner}a_2 = root(a_2) \wedge c_1^{\urcorner} \cdot {}^{\ulcorner}a_2 = 0$. Hence, $c_1 \circledast c_2$.

The rest follows from $a_1 \circledcirc a_2$ and it remains to show $((root(a_1) \mapsto root(a_3))|b_1 + c_1) \circledcirc a_3$. This can be calculated by similar considerations as above using $a_1 \circledast a_3$. Therefore, $((root(a_1) \mapsto root(a_3))|b_1) + a_{L-l} \in (P(\mathsf{i}) \circledcirc R(\mathsf{j}, \mathsf{i}.l)) \circledcirc Q$. □

The conjuncts $\mathsf{i}.l = \square$ in the middle and right inference rules are useful, since they show that the assignments involved do not introduce any memory leaks. To provide more intuition of what is happening in the leftmost rule of Lemma 5.6.1, we depicted the shapes of the trees in the pre- and postcondition in Figure 5.5.

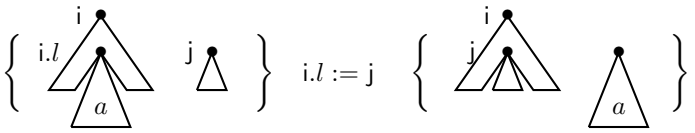


Figure 5.5: Illustration of a selector assignment inference rule.

Note that after the execution of selector assignment the subtree a still resides untouched on the heap. However, unless there are links to it from elsewhere, it is inaccessible and hence garbage. The other rules can be illustrated similarly.

5.6.2 Frame Rules

As a next step we introduce frame rules involving the newly introduced operators. Moreover, we provide validity proofs of them in an algebraic fashion. For this we mainly follow the relational treatment of Section 4.3. Concretely, the $\circledcirc$ and $\circledast$ operators are lifted to relations by

$$\begin{aligned}(s,a)\ C \circ D\ (s',a') \Leftrightarrow \exists\, a_1, a_2, a_1', a_2' : a = a_1 + a_2 \wedge a_1 \# a_2 \wedge a' = a_1' + a_2' \\ \wedge\ a_1' \# a_2' \wedge (s,a_1)\ C\ (s',a_1') \wedge (s,a_2)\ D\ (s',a_2')\,,\end{aligned}$$

where $\circ \in \{\circledcirc, \circledast\}$ and $\# \in \{\circledast, \triangleright\}$, respectively. Note that this fits into a treatment within multi-unit separation algebras as provided in Section 3.3 by adequate definitions of $\#$ on states rather than access elements. Cancellativity is ensured since both combinability relations $\circledast, \triangleright$ involve domain disjointness and $+$ denotes an abstract form of union on access elements.

For validating the frame rules we need in particular to ensure according to Theorem 4.3.28 that the new commands on access relations satisfy a corresponding version

of the frame property (cf. Definition 4.3.16) which we can use with the compensator H (cf. Definition 4.3.11), i.e.,

$$(\mathsf{safe}\,(C) \times \neg\bot)\,;\rhd\,;\,C \subseteq (C \times H)\,;\rhd\,.$$

Lemma 5.6.2 *All commands of Definition 5.5.11 have the frame property w.r.t.* $\circledast$.

Proof. The cases for allocation and deletion are not difficult. For simple variable assignments, only the store component is modified and the argumentation is the same as in our treatment for standard separation logic. Therefore we now concentrate on the selector assignment $\mathsf{i}.l := e$ for commands C. For the reader's benefit we repeat its non-aborting relational semantics:

$$\{\,((s,a),(s,(s(\mathsf{i}) \mapsto [\![\,e\,]\!]_{(s,a)})|a_l \,+\, a_{L-l}) : \mathsf{i} \in \mathit{dom}(s)\,,\; s(\mathsf{i}) \neq \Box\,,\; s(\mathsf{i}) \leq \ulcorner a_l\,\}\,.$$

First assume a non-aborting execution $((s,a),(s,a'_l + a_{L-l}))$ of C where $a'_l =_{df} (s(\mathsf{i}) \mapsto [\![\,e\,]\!]_{(s,a)})|a_l$, $[\![\,e\,]\!]_{(s,a)}$ is atomic and the above conditions are satisfied. Moreover we assume access elements with $a = a_p + a_r$ and $a_p \circledast a_r$ where for a store s we have that (s,a_p) is safe for C. By this, there exists a transition $((s,a_p),(s,b_p))$ of C where $b_p =_{df} (s(\mathsf{i}) \mapsto [\![\,e\,]\!]_{(s,a_p)})|(a_p)_l + (a_p)_{L-l}$ with $s(\mathsf{i}) \leq \ulcorner (a_p)_l$.

First, we show $b_p \circledast a_r$. By bilinearity of $\circledast$ we have

$$b_p \circledast a_r \;\Leftrightarrow\; (s(\mathsf{i}) \mapsto [\![\,e\,]\!]_{(s,a_p)})|(a_p)_l \circledast a_r \;\wedge\; (a_p)_{L-l} \circledast a_r\,.$$

The second conjunct follows by downward closedness of $\circledast$ from the assumption $a_p \circledast a_r$. The first conjunct is by Lemma 5.5.6 equivalent to $\overline{\ulcorner (s(\mathsf{i}) + [\![\,e\,]\!]_{(s,a_p)})} \cdot \overline{a_r} \leq \Box \wedge \overline{(\neg s(\mathsf{i}) \cdot (a_p)_l)} \cdot \overline{a_r} \leq \Box$. Again the latter conjunct follows from downward closedness of $a_p \circledast a_r$. For the former conjunct we first calculate $s(\mathsf{i}) \cdot \overline{a_r} \leq \ulcorner a_p \cdot \overline{a_r} \leq 0$ by $a_p \circledast a_r$. By Definition 5.5.1 we infer by isotony that $[\![\,e\,]\!]_{(s,a_p)} \leq [\![\,e\,]\!]_{(s,a)}$. Since $[\![\,e\,]\!]_{(s,a)}$ is atomic both tests are equal. Hence, by the assumptions and $a_p \circledast a_r$ we conclude that $[\![\,e\,]\!]_{(s,a_p)} \cdot \overline{a_r} \leq \Box$.

Finally, we need to show $a'_l + a_{L-l} = b_p + a_r$. Since $\ulcorner a_p \cdot \ulcorner a_r \leq 0$ and $s(\mathsf{i}) \leq \ulcorner a_p$ we have $(s(\mathsf{i}) \mapsto [\![\,e\,]\!]_{(s,a_p)})|(a_p)_l + (a_r)_l = (s(\mathsf{i}) \mapsto [\![\,e\,]\!]_{(s,a_p)})|a_l = (s(\mathsf{i}) \mapsto [\![\,e\,]\!]_{(s,a)})|a_l$. Moreover, $a_{L-l} = (a_p)_{L-l} + (a_r)_{L-l}$ and the claim follows. □

Corollary 5.6.3 *The* $\circledast$*-frame rule, i.e.,*

$$\frac{\{P\}\,C\,\{Q\}}{\{P \circledast R\}\,C\,\{Q \circledast R\}}$$

is valid for all predicates R and commands C that do not modify or reference any identifier occurring in R.

As a next step we give frame rules involving the operator $\circledcirc$. The usage of this operation requires the inclusion of predicates that characterise forests or tree-like structures as its combinability relation $\triangleright$ involves the definition of *root*. For simplicity we stay with tree structures as the rule is only required in later verification examples for trees. However, the frame rule can also be extended to forests like the symmetric inference rule we provide afterwards. By restricting ourselves to trees we only need a restricted version of frame property, i.e.,

$$(\mathsf{tree}\,;\mathsf{safe}\,(C)\;\times\;\mathsf{tree}\,;\neg\bot)\,;\rhd\;;\;C\;\subseteq\;(C\times H)\,;\rhd\,.$$

We call it the tree *-frame property w.r.t.* $\triangleright$. By this we conclude:

Lemma 5.6.4 *Any command C of Definition 5.5.11 has the* tree *-frame property w.r.t.* $\triangleright$.

Proof. The proof is similar as for Lemma 5.6.2. As before we consider only the case of selector assignments and assume a non-aborting execution $((s,a),(s,a'_l+a_{L-l}))$ of C where $a'_l =_{df} (s(\mathsf{i})\mapsto [\![\,e\,]\!]_{(s,a)})|a_l$ with $[\![\,e\,]\!]_{(s,a)}$ is atomic, $\mathsf{i}\in dom(s)$, $s(\mathsf{i})\neq\Box$ and $s(\mathsf{i})\leq\lceil a_l$. Moreover we assume trees a_p, a_r with $a=a_p+a_r$ and $a_p\triangleright a_r$ where for a store s we have that (s,a_p) is safe for C. By this, there exists a transition $((s,a_p),(s,b_p))$ of C where $b_p =_{df} (s(\mathsf{i})\mapsto [\![\,e\,]\!]_{(s,a_p)})|(a_p)_l+(a_p)_{L-l}$ with $s(\mathsf{i})\leq\lceil(a_p)_l$ and atomic test $[\![\,e\,]\!]_{(s,a_p)}$.

By this, $a_p\triangleright a_r$ implies $s(\mathsf{i})\cdot\overline{\lceil a_r}\rceil\leq 0\wedge[\![\,e\,]\!]_{(s,a_p)}\cdot a_r\rceil\leq\Box\wedge[\![\,e\,]\!]_{(s,a_p)}\cdot\lceil a_r\leq root(a_r)$. In particular, we calculate $root(a_r)=a_p\rceil\cdot\lceil a_r\leq a_p\rceil=(a_p)_l\rceil+(a_p)_{L-l}\rceil$ and since by assumption $s(\mathsf{i})\leq\lceil(a_p)_l$ and $s(\mathsf{i})\cdot root(a_r)\leq 0$ we can infer $root(a_r)\leq(\neg s(\mathsf{i})\cdot(a_p)_l)\rceil+(a_p)_{L-l}\rceil$. With these assumptions it is not difficult to calculate $b_p\triangleright a_r$. Moreover the rest of the claim follows by similar argumentations as in the proof of Lemma 5.6.3. □

Intuitively, the calculated inequation $root(a_r)\leq(\neg s(\mathsf{i})\cdot(a_p)_l)\rceil+(a_p)_{L-l}\rceil$ describes that the root of the tree a_r either remains unmodified in $(a_p)_l$ or is reachable via another selector $\neq l$ anyway. This describes in particular the local behaviour of the selector assignments.

Corollary 5.6.5 *Assume predicates $P,R\subseteq\mathsf{tree}$. The $\circledcirc$ -frame rule, i.e.,*

$$\frac{\{P\}\,C\,\{Q\}}{\{P\circledcirc R\}\,C\,\{Q\circledcirc R\}}$$

is valid for all predicates Q and commands C that do not modify any expression occurring in R and reference at most the roots of the trees in R.

Finally, we turn to a last variant of the frame rule. It is defined dually to Corollary 5.6.5 in the sense that the trees of the pre- and postcondition is appended to a leaf of the untouched trees characterised by R rather the other way round. For this we define a frame property restricted to forests by

$$(\mathsf{forest}\,;\neg\bot\,\times\,\mathsf{forest}\,;\,\mathsf{safe}\,(C))\,;\rhd\,;\,C\,\subseteq\,(H\times C)\,;\rhd\,.$$

We call it the *symmetric* forest *- frame property w.r.t.* $\rhd$. Clearly, a proof of the frame rule in this case requires a dual variant of the preservation property. Finally, we conclude:

Lemma 5.6.6 *Any command C of Definition 5.5.11 has the symmetric* forest *- frame property w.r.t.* $\rhd$.

Proof. The proof is similar as for Lemma 5.6.4. In the case of node deletion, the removal of roots connecting different trees is not allowed. We consider in he sequel only the case of selector assignments and assume a non-aborting execution $((s,a),(s,a'_l+a_{L-l}))$ of C where $a'_l =_{df} (s(\mathsf{i})\mapsto [\![\,e\,]\!]_{(s,a)})|a_l$ with $[\![\,e\,]\!]_{(s,a)}$ is atomic, $\mathsf{i}\in dom(s)$, $s(\mathsf{i})\neq\Box$ and $s(\mathsf{i})\leq\ulcorner a_l$. Moreover we assume forests a_p, a_r with $a=a_p+a_r$ and $a_r\rhd a_p$ where for a store s we have that (s,a_p) is safe for C. By this, there exists a transition $((s,a_p),(s,b_p))$ of C where $b_p =_{df} (s(\mathsf{i})\mapsto [\![\,e\,]\!]_{(s,a_p)})|(a_p)_l+(a_p)_{L-l}$ with $s(\mathsf{i})\leq\ulcorner(a_p)_l$ and atomic test $[\![\,e\,]\!]_{(s,a_p)}$.

By $a_r\rhd a_p$ the command C either modifies a trees $t_p =_{df} (a_p)_j$ for which there exists another tree $t_r =_{df} (a_r)_i$ with $t_r\rhd t_p$ or C modifies a disjoint tree t_p with $t_p\,\#\,t_r$ for arbitrary trees $t_r\leq a_r$. For the latter case we can use a similar argumentation as in Lemma 5.6.2 while for the former we again set $c|(t_p)_l+(t_p)_{L-l}\leq b_p$ with $c=_{df}(s(\mathsf{i})\mapsto [\![\,e\,]\!]_{(s,a_p)})$ and $s(\mathsf{i})\leq t_p$. By this and $[\![\,e\,]\!]_{(s,a_p)}\leq a_p^{\urcorner}$, we immediately infer with $a_r\rhd a_p$ that $\ulcorner t_r\cdot\overline{c}\leq 0\wedge t_r^{\urcorner}\cdot[\![\,e\,]\!]_{(s,a_p)}\leq\Box$. Moreover, by $t_r\rhd t_p$ we get $t_r^{\urcorner}\cdot s(\mathsf{i})\leq root(t_p)$. Since selector assignments do not delete nodes we have $\ulcorner t_p=\ulcorner(c|(t_p)_l+(t_p)_{L-l})$. With these assumptions it is not difficult any more to show $t_r\rhd(c|(t_p)_l+(t_p)_{L-l})$ assuming $t_r\rhd t_p$. □

Corollary 5.6.7 *Assume predicates $P,R\subseteq$* forest*. The symmetric* $\circledcirc$ *- frame rule, i.e.,*

$$\frac{\{P\}\,C\,\{Q\}}{\{R\circledcirc P\}\,C\,\{R\circledcirc Q\}}\,.$$

is valid for all predicates $R\subseteq$ forest *and commands C that do not modify or reference any expression occurring in R and do not delete the root of any tree in P.*

5.7 Verification Examples

For presenting our extended approach with the new operations and predicates in action we give some verification examples of concrete programs in the sequel.

5.7.1 List Reversal

This standard example is mainly intended to show the basic ideas of our approach. The algorithm is well known. It uses identifiers $\mathsf{i}, \mathsf{j}, \mathsf{k}$. The initial list is headed in i, while j heads the gradually accumulated result list. Finally, k is an auxiliary variable that remembers single list nodes while they are transferred from the original list to the result list:

$$\mathsf{j} := \square \;;\; \mathsf{while}\ (\mathsf{i} \neq \square)\ \mathsf{do}\ \big(\, \mathsf{k} := \mathsf{i.next} \;;\; \mathsf{i.next} := \mathsf{j} \;;\; \mathsf{j} := \mathsf{i} \;;\; \mathsf{i} := \mathsf{k} \,\big)\,.$$

To prove functional correctness of the in-situ reversal algorithm we introduce the concept of *abstraction functions* [Hoa72], e.g., to state invariant properties.

Definition 5.7.1 (Abstraction function for lists)
Assume $a \in \mathsf{list}$ and an atom $p \in \ulcorner a \urcorner$. We define the *abstraction function* li_a w.r.t. a which collects the nodes of the sublist of a starting in node p in a word consisting of these nodes in traversal order. Moreover, we define the semantics of the expression $\mathsf{i}^{\rightarrow}$ for a program identifier i as follows:

$$li_a(p) =_{df} \begin{cases} \langle\rangle & \text{if } p \cdot \ulcorner a \leq \square \\ \langle p \rangle \bullet li_a(\langle a|p) & \text{otherwise}\,, \end{cases} \qquad [\![\, \mathsf{i}^{\rightarrow} \,]\!]_{(s,a)} =_{df} li_a(s(\mathsf{i}))\,. \tag{5.1}$$

Here $\bullet$ stands for concatenation of words, $\langle\rangle$ denotes the empty word and $\langle p \rangle$ represents an atomic test p as a word.

Now using Hoare logic proof rules for variable assignment and while-loops (cf. Figure 2.3), we can provide a proof of the in-situ list reversal algorithm showing preservation of structural properties and functional correctness. As our invariant predicate for functional correctness of the algorithm we assume a word α and syntactically define $\mathrm{I} \Leftrightarrow_{df} (\mathsf{j}^{\rightarrow})^{\dagger} \bullet \mathsf{i}^{\rightarrow} = \alpha$, where $\_^{\dagger}$ denotes word reversal. By this the state-based semantics of I is given as

$$(s,a) \in \mathrm{I} \;\Leftrightarrow_{df}\; ([\![\, \mathsf{j}^{\rightarrow} \,]\!]_{(s,a)})^{\dagger} \bullet [\![\, \mathsf{i}^{\rightarrow} \,]\!]_{(s,a)} = \alpha \wedge \mathsf{i}, \mathsf{j} \in dom(s)\,,$$

where α represents a word. Clearly, this lifts immediately to tests. The correctness proof for this example can be found in Figure 5.6 assuming the selector set $L = \{\mathsf{next}\}$.

$$
\begin{array}{l}
\{\ \mathsf{list}\,(\mathsf{i}) \wedge \mathsf{i}^{\rightarrow} = \alpha\ \} \\
\ \mathsf{j} := \square\,; \\
\{\ \mathsf{list}\,(\mathsf{i}) \circledast \mathsf{list}\,(\mathsf{j}) \wedge \mathrm{I}\ \} \\
\ \mathsf{while}\ (\mathsf{i} \neq \square)\ \mathsf{do}\ (\\
\quad \{\ (\mathsf{cell}\,(\mathsf{i}) \circledcirc \mathsf{list})\ \circledast \mathsf{list}\,(\mathsf{j}) \wedge \mathrm{I}\ \} \\
\quad \mathsf{k} := \mathsf{i.next}; \\
\quad \{\ (\mathsf{cell}\,(\mathsf{i}) \circledcirc \mathsf{list}\,(\mathsf{k}))\ \circledast \mathsf{list}\,(\mathsf{j}) \wedge (\mathsf{j}^{\rightarrow})^{\dagger} \bullet \mathsf{i} \bullet \mathsf{k}^{\rightarrow} = \alpha\ \} \\
\quad \{\ (\mathsf{cell}\,(\mathsf{i}) \circledcirc \mathsf{list}\,(\mathsf{k}))\ \circledast \mathsf{list}\,(\mathsf{j}) \wedge (\mathsf{i} \bullet \mathsf{j}^{\rightarrow})^{\dagger} \bullet \mathsf{k}^{\rightarrow} = \alpha\ \} \\
\quad \mathsf{i.next} := \mathsf{j}\,; \\
\quad \{\ (\mathsf{cell}\,(\mathsf{i}) \circledcirc \mathsf{list}\,(\mathsf{j}))\ \circledast \mathsf{list}\,(\mathsf{k}) \wedge (\mathsf{i} \bullet \mathsf{j}^{\rightarrow})^{\dagger} \bullet \mathsf{k}^{\rightarrow} = \alpha\ \} \\
\quad \{\ \mathsf{list}\,(\mathsf{i}) \circledast \mathsf{list}\,(\mathsf{k}) \wedge (\mathsf{i}^{\rightarrow})^{\dagger} \bullet \mathsf{k}^{\rightarrow} = \alpha\ \} \\
\quad \mathsf{j} := \mathsf{i}\,;\ \mathsf{i} := \mathsf{k}\,; \\
\quad \{\ \mathsf{list}\,(\mathsf{j}) \circledast \mathsf{list}\,(\mathsf{i}) \wedge \mathrm{I}\ \} \\
)\ \\
\{\ \mathsf{list}\,(\mathsf{j}) \wedge (\mathsf{j}^{\rightarrow})^{\dagger} = \alpha\ \} \\
\{\ \mathsf{list}\,(\mathsf{j}) \wedge \mathsf{j}^{\rightarrow} = \alpha^{\dagger}\ \}
\end{array}
$$

Figure 5.6: Verification of list reversal.

It can be seen that each assertion consists of a part that abstracts the structure of the considered data structure and another part that connects the concrete and abstract levels of reasoning. The same pattern will also occur in the example algorithms of the following sections.

Compared to the list reversal algorithm given in [Rey09] we hide in the $\circledcirc$ operator the existential quantifiers that were necessary for the standard approach of separation logic to describe the sharing relationships. Moreover, we include all correctness properties of the occurring data structures and their interrelationship in the definitions of the new connectives and predicates. To state functional correctness, no quantifiers are needed due to the use of the abstraction function. Hence the formulas become easier to read and more concise.

For a variant (inspired by [CS10]), if one would, e.g., exchange the first two commands in the while loop of the list reversal algorithm, one could possibly create a memory leak. It can be seen that after the assignment $\mathsf{i.next} := \mathsf{j}$ one would get in the postcondition as the structural part the formula $(\mathsf{cell}\,(\mathsf{i}) \circledcirc \mathsf{list}\,(\mathsf{j}))\ \circledast\ \mathsf{list}$. The list memory part separated out by the second argument of $\circledast$ can neither be reached from i nor from j. Moreover, there is no program variable containing a reference to the root of that part.

5.7.2 Tree Rotation

We now consider a more complex example. As already mentioned, for binary trees we use the selector names left and right. Therefore we set $L = \{\mathsf{left}, \mathsf{right}\}$ and $a =_{df} a_{\mathsf{left}} + a_{\mathsf{right}}$ for this section. To define an abstraction function $^{\leftrightarrow}$ similar to the $^{\rightarrow}$ function in Equation (5.1), we view abstract trees as being inductively defined: An *abstract tree* is either the empty tree, represented by the empty word $\langle\rangle$, or it is a triple $\langle T_l, \langle p\rangle, T_r\rangle$, consisting of an atomic test p as a word that represents the root node and further abstract trees T_l, T_r, the left and right subtrees, respectively. Again $\langle \dots \rangle$ represents a sequence of nodes by a corresponding word. By this we set

$$\begin{aligned} tr_a(p) &=_{df} \begin{cases} \langle\rangle & \text{if } p \cdot \ulcorner a \leq \square \\ \langle tr_a(\langle a_{\mathsf{left}}|p\rangle),\ \langle p\rangle,\ tr_a(\langle a_{\mathsf{right}}|p\rangle)\rangle & \text{otherwise}, \end{cases} \\ [\![\, \mathsf{i}^{\leftrightarrow} \,]\!]_{(s,a)} &=_{df} tr_a(s(\mathsf{i}))\,. \end{aligned} \tag{5.2}$$

Note that expressions $[\![\, \mathsf{i}^{\leftrightarrow} \,]\!]_{(s,a)}$ are only defined if $\mathsf{i} \in dom(s)$. Hence we define $(s,a) \in (\, \mathsf{i}^{\leftrightarrow} = \langle \dots \rangle\,) \Leftrightarrow_{df} [\![\, \mathsf{i}^{\leftrightarrow} \,]\!]_{(s,a)} = \langle \dots \rangle \wedge \mathsf{i} \in dom(s)$ similarly to the case of the invariant I on lists.

For a concrete example, we now present a correctness proof of an algorithm for tree rotation as known from the data structure of AVL trees. The algorithms starts with the left tree in Figure 5.7 and ends with the rotated one on the right.

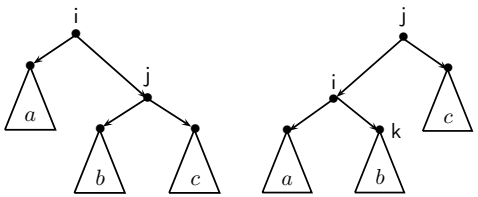


Figure 5.7: Tree rotation at the beginning and end.

Using our basic tree predicates a formula describing the shape of the left tree of Figure 5.7 would read

$$\mathsf{cell}\,(\mathsf{i}) \circledcirc (\mathsf{tree}(\mathsf{i.left}) \circledast (\mathsf{cell}\,(\mathsf{i.right}) \circledcirc (\mathsf{tree}(\mathsf{i.right.left}) \circledast \mathsf{tree}(\mathsf{i.right.right}))))\,. \tag{5.3}$$

Unfortunately, this formula is hard to read and difficult to understand. To overcome this issue we define some auxiliary predicates that will make the assertions easier to read and more concise. The resulting formulas will exactly describe the required components of the considered tree. Concretely for trees we set

$$\begin{aligned} \mathsf{lt\_context}(\mathsf{i}) &=_{df} \mathsf{cell}\,(\mathsf{i}) \circledcirc \mathsf{tree}(\mathsf{i.right})\,, \\ \mathsf{r\_tree}(\mathsf{i}) &=_{df} \mathsf{lt\_context} \cap (\mathsf{i.left} = \square)\,, \\ \mathsf{rt\_context}(\mathsf{i}) &=_{df} \mathsf{cell}\,(\mathsf{i}) \circledcirc \mathsf{tree}(\mathsf{i.left})\,, \\ \mathsf{l\_tree}(\mathsf{i}) &=_{df} \mathsf{rt\_context}(\mathsf{i}) \cap (\mathsf{i.right} = \square)\,. \end{aligned}$$

Intuitively, lt_context(i) describes a tree where i represents the root of the tree and by the selector right we reach another tree. The context on the left selector is not specified concretely. Symmetrically, rt_context(i) asserts the dual tree structure. The predicates r_tree(i) and l_tree(i) describe trees where the unspecified context is empty, i.e., the address of corresponding selector equals nil.

By this we can transform Formula (5.3) using Lemma 5.5.3(c) into

$$\mathsf{rt\_context(i)} \circledcirc (\mathsf{lt\_context(i.right)} \circledcirc \mathsf{tree(i.right.left)})\,. \tag{5.4}$$

A proof of this can be found in the Appendix. We now give a "clean" version of the tree rotation algorithm, in which all occurring subtrees are separated. After that we will show an optimised version, however, with sharing in an intermediate state. With the above new predicates, a correctness proof is given in Figure 5.8.

$\{\, \mathsf{rt\_context(i)} \circledcirc (\mathsf{lt\_context(i.right)} \circledcirc \mathsf{tree(i.right.left)}) \wedge \mathsf{i}^{\leftrightarrow} = \langle T_l, p, \langle T_k, q, T_r\rangle\rangle \,\}$
$\mathsf{j} := \mathsf{i.right}\,;$
$\{\, \mathsf{rt\_context(i)} \circledcirc (\mathsf{lt\_context(i.right, j)} \circledcirc \mathsf{tree(j.left)}) \wedge$
$\quad \mathsf{i}^{\leftrightarrow} = \langle T_l, p, \langle T_k, q, T_r\rangle\rangle \wedge \mathsf{j}^{\leftrightarrow} = \langle T_k, q, T_r\rangle \,\}$
$\{\, (\mathsf{rt\_context(i)} \circledcirc \mathsf{lt\_context(i.right, j)}) \circledcirc \mathsf{tree(j.left)} \wedge$
$\quad \mathsf{i}^{\leftrightarrow} = \langle T_l, p, \langle T_k, q, T_r\rangle\rangle \wedge \mathsf{j}^{\leftrightarrow} = \langle T_k, q, T_r\rangle \,\}$
$\mathsf{i.right} := \square\,;$
$\{\, (\mathsf{l\_tree(i)} \circledast \mathsf{lt\_context(j)}) \circledcirc \mathsf{tree(j.left)} \wedge \mathsf{i}^{\leftrightarrow} = \langle T_l, p, \langle\rangle\rangle \wedge \mathsf{j}^{\leftrightarrow} = \langle T_k, q, T_r\rangle \,\}$
$\{\, \mathsf{l\_tree(i)} \circledast (\mathsf{lt\_context(j)} \circledcirc \mathsf{tree(j.left)}) \wedge \mathsf{i}^{\leftrightarrow} = \langle T_l, p, \langle\rangle\rangle \wedge \mathsf{j}^{\leftrightarrow} = \langle T_k, q, T_r\rangle \,\}$
$\mathsf{k} := \mathsf{j.left}\,;$
$\{\, \mathsf{l\_tree(i)} \circledast (\mathsf{lt\_context(j)} \circledcirc \mathsf{tree(j.left, k)}) \wedge$
$\quad \mathsf{i}^{\leftrightarrow} = \langle T_l, p, \langle\rangle\rangle \wedge \mathsf{j}^{\leftrightarrow} = \langle T_k, q, T_r\rangle \wedge \mathsf{k}^{\leftrightarrow} = T_k \,\}$
$\mathsf{j.left} := \square\,;$
$\{\, \mathsf{l\_tree(i)} \circledast \mathsf{r\_tree(j)} \circledast \mathsf{tree(k)} \wedge \mathsf{i}^{\leftrightarrow} = \langle T_l, p, \langle\rangle\rangle \wedge \mathsf{j}^{\leftrightarrow} = \langle\langle\rangle, q, T_r\rangle \wedge \mathsf{k}^{\leftrightarrow} = T_k \,\}$
$\mathsf{j.left} := \mathsf{i}\,;$
$\{\, (\mathsf{lt\_context(j)} \circledcirc \mathsf{l\_tree(i, j.left)}) \circledast \mathsf{tree(k)} \wedge$
$\quad \mathsf{i}^{\leftrightarrow} = \langle T_l, p, \langle\rangle\rangle \wedge \mathsf{j}^{\leftrightarrow} = \langle\langle T_l, p, \langle\rangle\rangle, q, T_r\rangle \wedge \mathsf{k}^{\leftrightarrow} = T_k \,\}$
$\{\, \mathsf{lt\_context(j)} \circledcirc (\mathsf{l\_tree(i, j.left)} \circledast \mathsf{tree(k)}) \wedge$
$\quad \mathsf{i}^{\leftrightarrow} = \langle T_l, p, \langle\rangle\rangle \wedge \mathsf{j}^{\leftrightarrow} = \langle\langle T_l, p, \langle\rangle\rangle, q, T_r\rangle \wedge \mathsf{k}^{\leftrightarrow} = T_k \,\}$
$\mathsf{i.right} := \mathsf{k}\,;$
$\{\, \mathsf{lt\_context(j)} \circledcirc (\mathsf{rt\_context(i, j.left)} \circledcirc \mathsf{tree(k, i.right)}) \wedge$
$\quad \mathsf{j}^{\leftrightarrow} = \langle\langle T_l, p, T_k\rangle, q, T_r\rangle \wedge \mathsf{i}^{\leftrightarrow} = \langle T_l, p, T_k\rangle \wedge \mathsf{k}^{\leftrightarrow} = T_k \,\}$

Figure 5.8: Verification of tree rotation.

Note that the predicate $(\mathsf{i}.l = \square)$ satisfies the equation $(P(\mathsf{i}) \circledast Q) \cap (\mathsf{i}.l = \square) =$

$(P(\mathsf{i}) \cap (\mathsf{i}.l = \square)) \circledast Q$ for $P, Q \subseteq \mathsf{tree}$. Therefore we can use Lemma 5.6.1 for the proof. Moreover, the proof is structured into two parts. First, each assignment changes the structural part involving the tree and context predicates which guarantee implicitly the preservation of the tree structure and its required properties. Second, the assignments modify the conjunct that involves the abstraction functions for guaranteeing correctness of functional properties, e.g., preservation of values within the considered tree.

The next version of the algorithm that we present uses fewer assignments, but shows sharing within an intermediate state. Its verification requires the definition of a new predicate, since one of the intermediate states cannot be described with the operators we have defined so far.

Definition 5.7.2 (Sharing predicate)
For predicates $P, R \subseteq \mathsf{forest}$ and $Q \subseteq \mathsf{tree}$ we define

$$P \circledR Q \circledL R \;=_{df}\; \{\, a_1 + a_2 + a_3 : a_1 \in P,\, a_2 \in Q,\, a_3 \in R,\\ a_1 \triangleright a_2,\; a_3 \triangleright a_2,\; \overline{a_1} \cdot \overline{a_3} = root(a_2) \,\}\,.$$

Clearly, $P \circledR Q \circledL R = R \circledR Q \circledL P$. The linked structure characterised by the predicate is depicted in Figure 5.9.

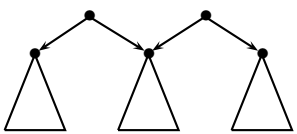


Figure 5.9: A shared subtree.

For a use of this predicate in concrete verifications we need to introduce the following inference rules.

Lemma 5.7.3 *Assume predicates* $P \subseteq \mathsf{lt\_context}$ *and* $Q, R \subseteq \mathsf{tree}$, *identifiers* i, j *and selectors* $l, m \in L$ *then*

$$\{\, (P(\mathsf{i}) \circledR (Q(\mathsf{j}, \mathsf{i}.l) \circledR R(\mathsf{j}.m)) \,\} \\ \mathsf{i}.l := \mathsf{j}.m\,; \\ \{\, P(\mathsf{i}) \circledR R(\mathsf{j}.m, \mathsf{i}.l) \circledL Q(\mathsf{j}) \,\}\,,$$

$$\{\, P(\mathsf{i}) \circledR S(\mathsf{j}.m, \mathsf{i}.l) \circledL R(\mathsf{j}) \,\} \\ \mathsf{i}.l := \mathsf{j}\,; \\ \{\, P(\mathsf{i}) \circledR (R(\mathsf{j}, \mathsf{i}.l) \circledR S(\mathsf{j}.m)) \,\}\,.$$

The latter rule also works for $P \subseteq \mathsf{tree}$.

Proof. We outline a proof of the first rule; a proof for the second one can be obtained similarly. Assume a non-aborting execution involving the tree $a = a_1 + a_2 + a_3$

with $a_i \in P(\mathsf{i}) \wedge a_2 \in Q(\mathsf{j},\mathsf{i}.l) \wedge a_3 \in R(\mathsf{j}.m)$. We know $a_1 \triangleright (a_2 + a_3) \wedge a_2 \triangleright a_3$ and from the identifiers $(s(\mathsf{i}) \mapsto [\![\mathsf{j}.m]\!]_{(s,a)}) = (root(a_1) \mapsto root(a_3))$. Note that $a_1 \in P(\mathsf{i})$ immediately implies $(root(a_1) \mapsto root(a_3))|(a_1)_l = (root(a_1) \mapsto root(a_3))$. Next we set $b_1 =_{df} (root(a_1) \mapsto root(a_3)) + (a_1)_{L-l}$. From the assumption we get $(a_1)_{L-l} \circledast a_2$. Using Lemma 5.4.12, we also know $a_1 \circledast a_3$ and further infer $(a_1)_{L-l} \circledast a_3$. Now we can conclude $b_1 \triangleright a_3 \wedge \overline{b_1} \cdot \overline{a_2} = root(a_3)$. □

By Lemma 5.7.3 we can verify a shorter form of the tree rotation algorithm that uses sharing which can be found in Figure 5.10.

$\{\ \mathsf{rt\_context}(\mathsf{i}) \circledcirc (\mathsf{lt\_context}(\mathsf{i.right}) \circledcirc \mathsf{tree}(\mathsf{i.right.left})) \wedge \mathsf{i}^{\leftrightarrow} = \langle T_l, p, \langle T_k, q, T_r\rangle\rangle\ \}$
$\mathsf{j} := \mathsf{i.right}\ ;$
$\{\ \mathsf{rt\_context}(\mathsf{i}) \circledcirc (\mathsf{lt\_context}(\mathsf{i.right},\mathsf{j}) \circledcirc \mathsf{tree}(\mathsf{j.left})) \wedge$
$\mathsf{i}^{\leftrightarrow} = \langle T_l, p, \langle T_k, q, T_r\rangle\rangle \wedge \mathsf{j}^{\leftrightarrow} = \langle T_k, q, T_r\rangle\ \}$
$\mathsf{i.right} := \mathsf{j.left}\ ;$
$\{\ \mathsf{rt\_context}(\mathsf{i}) \circledcirc \mathsf{tree}(\mathsf{j.left},\mathsf{i.right}) \circledast \mathsf{lt\_context}(\mathsf{j}) \wedge$
$\mathsf{i}^{\leftrightarrow} = \langle T_l, p, T_k\rangle \wedge \mathsf{j}^{\leftrightarrow} = \langle T_k, q, T_r\rangle\ \}$
$\mathsf{j.left} := \mathsf{i}\ ;$
$\{\ \mathsf{lt\_context}(\mathsf{j}) \circledcirc (\mathsf{rt\_context}(\mathsf{i},\mathsf{j.left}) \circledcirc \mathsf{tree}(\mathsf{i.right})) \wedge$
$\mathsf{j}^{\leftrightarrow} = \langle\langle T_l, p, T_k\rangle, q, T_r\rangle \wedge \mathsf{i}^{\leftrightarrow} = \langle T_l, p, T_k\rangle \wedge \mathsf{k}^{\leftrightarrow} = T_k\ \}$

Figure 5.10: Tree rotation with sharing.

The third assertion, that uses the new predicate, can be depicted as in Figure 5.11. It represents the situation where one subtree is shared within two trees in an intermediate state.

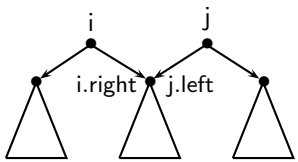


Figure 5.11: Depiction of the intermediate state with sharing.

5.8 A Treatment of Overlaid Data Structures

We continue to further underpin the practicality of our approach. For this we consider as a further concrete example the treatment of overlaid data structures with so-called *threaded trees*. These trees enable through their threads a fast inorder traversal of the whole tree (cf. Figure 5.12, where the dashed lines denote threads). This is done by saving references to the next node of the traversal sequence in previously unused space. In the case of Figure 5.12 threads are stored in the right-link. For an inorder traversal one needs to get to the leftmost node j from the root i. The root and the nodes on its left and right-selector are not marked, i.e., from that nodes one has to follow the right-link to the right node and further move to the leftmost node from there to get to the subsequent node in an inorder traversal.

For a formal treatment of that data structure first note that all predicates and operations defined up to now consider non-reachability or directed reachability only on complete access elements, i.e., the operators work on all selectors. This is far too strict, especially in the case of threaded trees. As an example, $\triangleright$ completely excludes the existence of cycles in the whole tree while e.g., links of the tree and threads of the list together might form cycles within such a tree. In Figure 5.12 we can directly reach a cycle from j to its successor via the thread and back via the left selector.

Hence, we need a weaker variant of $\triangleright$ that works on a specific set of links $M \subseteq L$. For a linked structure c over L we set $c_M =_{df} \sum_{l \in M} c_l$ and define

$$a \triangleright_M b \Leftrightarrow_{df} a_M \triangleright b_M$$

and its corresponding operator on predicates by

$$P \circledcirc_M Q =_{df} \{ a + b : a \in P, b \in Q, a \triangleright_M b \} .$$

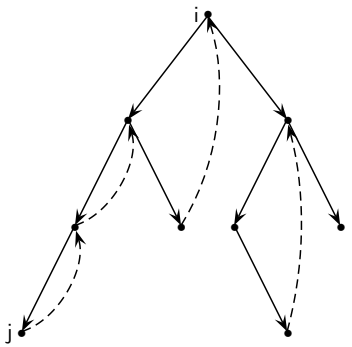


Figure 5.12: A threaded tree.

We will omit the set braces when M is a singleton set. The same generalisations can be apply to $\circledast$ and $\circledcirc$. Note that, by $M \subseteq L$ and downward closedness of $\circledcirc$, also $\circledcirc \subseteq \circledcirc_M$ and hence $P \circledast Q \subseteq P \circledast_M Q$. Note also that our laws for $\circledcirc$ and $\circledast$ hold also for $\circledcirc_M$ and $\circledast_M$, respectively, assuming a set of links $M \subseteq L$.

For a threaded tree we define the access element by $a = a_{\mathsf{left}} + a_{\mathsf{right}} + a_{\mathsf{marked}}$, i.e., $L = \{\mathsf{left}, \mathsf{right}, \mathsf{marked}\}$. Clearly, the access elements a_{left} and a_{right} need to be disjoint, while a_{marked} is a test with $a_{\mathsf{marked}} \leq \lceil a_{\mathsf{right}}$. It represents a set of nodes from which threads emanate, i.e., where the right links represent pointers from the respective nodes to their successor in the inorder traversal of the corresponding unthreaded tree.

In a "real" program this test would be implemented by marking bits on the nodes; it is here treated as an access element for uniformity.

Based on this, we define a *virtual access element* a_{thread}; this means that the selector thread is not contained in L, but the element is constructed using selectors of L. It reflects the fact that in a threaded tree the right-links can be of two kinds: at marked nodes they are "regular" links to non-empty right subtrees, whereas at non-marked nodes they are "thread" links that point back to ancestor nodes:

$$\begin{aligned} a_{\mathsf{thread}} &=_{df} a_{\mathsf{marked}} \cdot a_{\mathsf{right}} + a_{\mathsf{RLm}}\,, \\ a_{\mathsf{RLm}} &=_{df} (\neg a_{\mathsf{marked}} \cdot a_{\mathsf{right}}) \cdot a_{\mathsf{left}}^{*} \cdot \neg\lceil a_{\mathsf{left}}\,. \end{aligned}$$

In addition, we require the following structural properties of a:

(a) $a_{\mathsf{LR}} =_{df} a_{\mathsf{left}} + \neg a_{\mathsf{marked}} \cdot a_{\mathsf{right}}$ forms a tree,

(b) a_{thread} forms a chain,

(c) the inorder sequence of a_{LR} equals the traversal sequence of a_{thread}.

The access element a_{RLm} connects a non-marked node x, i.e., a node without any threads, with the leftmost node in the right subtree of x which denotes its successor node in the inorder traversal. The subexpression $a_{\mathsf{left}}^{*} \cdot \neg\lceil a_{\mathsf{left}}$ occurring in a_{RLm} is an algebraic representation of the loop while $\lceil a_{\mathsf{left}}$ do a_{left}. It has been shown in [DM01b] that determinacy of a loop body is inherited by the corresponding while loop. Note that a_{thread} is a virtual access relation, i.e., its selector thread is not contained in L, but its access relation is formed using selectors of L.

Next, we relax the definition for some predicates, so that they take the new linked structures into account:

$$\begin{aligned} \mathsf{u\_cell} &=_{df} \{a : a_{\mathsf{LR}} \text{ is a cell, } a_{\mathsf{marked}} \leq 0\}\,, \\ \mathsf{m\_cell} &=_{df} \{a : a_{\mathsf{LR}} \text{ is a cell, } a_{\mathsf{marked}} = root(a)\}\,, \\ \mathsf{thread\_list} &=_{df} \{a : a_{\mathsf{thread}} \text{ is a chain}\}\,, \\ \mathsf{lr\_tree} &=_{df} \{a : a_{\mathsf{LR}} \text{ is a tree}\}\,. \end{aligned}$$

The predicate u_cell characterises unmarked cells while nodes in m_cell are marked. Marking is realised by assigning the root of a cell to the marked component. The effect of this is that the behaviour of a Boolean value is mimicked. Moreover, the predicate thread_list is restricted to all marked right selectors and connections from unmarked nodes to left-most nodes while lr_tree considers only the left and unmarked right selectors. We further define

$$[\![\mathsf{j}^{\rightarrow}]\!]_{(s,a)} =_{df} li_{a_{\mathsf{thread}}}(s(\mathsf{j})) \quad \text{and} \quad [\![\mathsf{i}^{\rightsquigarrow}]\!]_{(s,a)} =_{df} inorder(tr_{a_{\mathsf{LR}}}(s(\mathsf{i})))\,, \qquad (5.5)$$

where $\mathsf{i},\mathsf{j} \in dom(s)$ and $tr_a(p)$ for a tree a is defined in Equation (5.2). The function $inorder(\langle \ldots \rangle)$ returns the word consisting of the nodes $\langle p \rangle$ of the considered tree in the sequence of an inorder traversal, i.e., it deletes in particular all $\langle$ and $\rangle$ that are only used for the tree structure. A threaded tree is now defined by the predicate

$$\mathsf{th\_tree}(\mathsf{i},\mathsf{j}) \;=_{df}\; \mathsf{lr\_tree}\,(\mathsf{i}) \wedge \mathsf{thread\_list}\,(\mathsf{j}) \wedge \mathsf{j}^{\rightarrow} = \mathsf{i}^{\rightsquigarrow}\,,$$

where i points to the root of the underlying tree and j points to the head of the list formed by a_{thread} (cf. Figure 5.12). Note that $\mathsf{j}^{\rightarrow} = \mathsf{i}^{\rightsquigarrow}$ implies that $\mathsf{j} = \mathsf{leftmost}(\mathsf{i})$, where

$$lm_a(p) \;=_{df}\; \begin{cases} \square & \text{if } p = \square \\ p & \text{if } (\langle a_{\mathsf{left}}|p) \cdot \ulcorner a = 0 \\ lm_a(\langle a_{\mathsf{left}}|p) & \text{otherwise}\,, \end{cases} \qquad [\![\, \mathsf{leftmost}(\mathsf{i}) \,]\!]_{(s,a)} \;=_{df}\; lm_a(s(\mathsf{i}))\,.$$

Before we continue with the verification example we need to sum up a few consequences of these definitions.

Lemma 5.8.1 *Assume predicates $P, Q \subseteq \mathsf{tree}$ and identifiers i,j. Moreover assume selector sets $K, M \subseteq L$ and a selector $l \in K - M$. Then*

$$\begin{array}{l} \{\; P(\mathsf{i}) \circledast_K Q(\mathsf{j}) \;\} \\ \;\mathsf{i}.l := \mathsf{j}\,; \\ \{\; P(\mathsf{i}) \circledast_{K-l} Q(\mathsf{j}) \wedge P(\mathsf{i}) \circledcirc_l Q(\mathsf{j}, \mathsf{i}.l) \;\}\,, \end{array}$$

$$\begin{array}{l} \{\; P(\mathsf{i}) \circledcirc Q(\mathsf{j}) \wedge \mathsf{i}.l = \square \;\} \\ \;\mathsf{i}.l := \mathsf{j}\,; \\ \{\; P(\mathsf{i}) \circledcirc Q(\mathsf{j}) \wedge P(\mathsf{i}) \circledcirc_l Q(\mathsf{j}, \mathsf{i}.l) \;\} \end{array} \quad \textit{and} \quad \begin{array}{l} \{\; P(\mathsf{i}) \circledcirc_M Q(\mathsf{j}) \wedge \mathsf{j}.l = \square \;\} \\ \;\mathsf{j}.l := \mathsf{i}\,; \\ \{\; P(\mathsf{i}) \circledcirc_M Q(\mathsf{j}) \wedge Q(\mathsf{j}) \circledcirc_l P(\mathsf{i}, \mathsf{j}.l) \;\}\,. \end{array}$$

Proofs for these rules can be constructed similarly to that of Lemma 5.6.1. All of these inference rules make use of the generalised operators. The first rule describes that after the selector assignment P and Q remain strongly disjoint on all selectors in $K - l$ while it is now possible to reach Q from P via the selector l. This is similarly mimicked in the second rule. It describes that Q is reachable from P; especially one can use the selector l to reach Q from P after the execution of the selector assignment command. The third rule describes that all links from P to Q mentioned in the precondition will remain unchanged by assignments via a selector $l \notin M$. Note that these rules also extend to forests but suffice in this form for the subsequent example.

Next, we consider marking of nodes. For that we define a command that appropriately sets the marked selector of the considered access elements and redefines allocation of

nodes to ignore the mark selector:

$$\begin{aligned} \mathsf{mark(i)} &=_{df} \{((s,a),(s,(s(\mathsf{i})+a_{\mathsf{marked}})+a_{L-\mathsf{marked}})) : \mathsf{i} \in dom(s)\}, \\ \mathsf{i} := \mathsf{new\,cell}\,() &=_{df} \{((s,a),(s[\mathsf{i} \leftarrow p],(p \mapsto \square)|a_{L-\mathsf{marked}}+a_{\mathsf{marked}}) : \mathsf{i} \in dom(s), \\ & \qquad p \text{ is an atomic test}, p \leq \neg\ulcorner a, p \neq \square\}. \end{aligned}$$

Note that both commands satisfy the frame rules of Section 5.6.2. Before we can use them in the verification of the concrete example we give further inference rules.

Lemma 5.8.2 *Assume identifiers* $\mathsf{i},\mathsf{j},\mathsf{k}$ *with* $\mathsf{i},\mathsf{k} \neq \square$ *and a word* α *then*

$$\begin{array}{ll} \{\ \mathsf{th\_tree(i,j)} \circledast \mathsf{u\_cell\,(k)}\ \} & \{\ \mathsf{u\_cell\,(k)}\ \} \\ \mathsf{j.left} := \mathsf{k}\,; & \mathsf{mark(k)}\,; \\ \{\ (\mathsf{lr\_tree\,(i)} \circledcirc_{\mathsf{LR}} \mathsf{u\_cell\,(k)}) \wedge \mathsf{thread\_list\,(j)} \wedge \mathsf{k} \bullet \mathsf{j}^{\rightarrow} = \mathsf{i}^{\rightsquigarrow}\ \}, & \{\ \mathsf{m\_cell\,(k)}\ \}, \end{array}$$

$$\begin{array}{ll} & \{\ (\mathsf{u\_cell\,(k)} \circledcirc_{\mathsf{right}} \mathsf{thread\_list\,(j,k.right)}) \wedge \mathsf{k} \bullet \mathsf{j}^{\rightarrow} = \alpha\ \} \\ \textit{and} & \mathsf{mark(k)}\,; \\ & \{\ (\mathsf{m\_cell\,(k)} \circledcirc_{\mathsf{thread}} \mathsf{thread\_list\,(j,k.right)}) \wedge \mathsf{k}^{\rightarrow} = \alpha\ \}. \end{array}$$

These laws are direct consequences of the definition of mark and the abstraction functions in Equation (5.5). Proofs can be given similarly as before. The first inference rule expresses that after making the unmarked cell in k the left subtree of j, the inorder list of the resulting overall tree now starts with k and continues with that headed by j. The meaning of the second rule is obvious. The third rule states that after the execution of mark the right-link of k represents a thread link, so that the thread list is now headed by k.

We can now give another verification example to view the new predicates and operators in action. For simplicity, we do not treat balancing so that we can simply add a new node as the left subtree of the leftmost node. We assume a non-empty threaded tree with root in i and $\mathsf{j} \neq \mathsf{i}$ heading the thread list. Then we can reason as in Figure 5.13.

Note that the abstraction functions on i, j are independent of the newly allocated cell k. Hence, as in the case of tree rotation we can infer e.g., $(P \cap (\mathsf{j}^{\rightarrow} = \mathsf{i}^{\rightsquigarrow})) \circledast \mathsf{u\_cell\,(k)} = (P \circledast \mathsf{u\_cell\,(k)}) \cap (\mathsf{j}^{\rightarrow} = \mathsf{i}^{\rightsquigarrow})$. Finally, we conclude this section by sketching a similar idea for treating doubly-linked lists. An adequate access element can be given by $a = a_{\mathsf{next}} + a_{\mathsf{prev}}$ with $L = \{\mathsf{next},\mathsf{prev}\}$. The characterising predicate for this data structure then reads

$$\mathsf{dl\_list(i,j)} =_{df} \mathsf{next\_list\,(i)} \wedge \mathsf{prev\_list\,(j)} \wedge \mathsf{i}^{\rightarrow} = (^{\leftarrow}\mathsf{j})^{\dagger},$$

where

$$\mathsf{next\_list} =_{df} \{a : a_{\mathsf{next}} \text{ is a chain}\}, \qquad \mathsf{prev\_list} =_{df} \{a : a_{\mathsf{prev}} \text{ is a chain}\}.$$

$\{\ \mathsf{lr\_tree}(\mathsf{i}) \circledcirc_{\mathsf{LR}} \mathsf{u\_cell}(\mathsf{j}) \ \wedge\ \mathsf{thread\_list}(\mathsf{j}) \ \wedge\ \mathsf{j}^{\rightarrow} = \mathsf{i}^{\rightsquigarrow} \ \wedge\ \mathsf{j}^{\rightarrow} = \alpha\ \}$
$\mathsf{k} := \mathsf{new\ cell}()\,;$
$\{\ (\mathsf{lr\_tree}(\mathsf{i}) \circledcirc_{\mathsf{LR}} \mathsf{u\_cell}(\mathsf{j})) \circledast \mathsf{u\_cell}(\mathsf{k}) \ \wedge\ \mathsf{thread\_list}(\mathsf{j}) \circledast \mathsf{u\_cell}(\mathsf{k})$
$\quad \wedge\ \mathsf{j}^{\rightarrow} = \mathsf{i}^{\rightsquigarrow} \ \wedge\ \mathsf{j}^{\rightarrow} = \alpha\ \}$
$\mathsf{j.left} := \mathsf{k}\,;$
$\{\ \mathsf{lr\_tree}(\mathsf{i}) \circledcirc_{\mathsf{LR}} (\mathsf{u\_cell}(\mathsf{j}) \circledcirc_{\mathsf{LR}} \mathsf{u\_cell}(\mathsf{k}, \mathsf{j.left})) \ \wedge\ \mathsf{thread\_list}(\mathsf{j}) \circledast_{\mathsf{right}} \mathsf{u\_cell}(\mathsf{k}, \mathsf{j.left})$
$\quad \wedge\ \mathsf{k} \bullet \mathsf{j}^{\rightarrow} = \mathsf{i}^{\rightsquigarrow} \ \wedge\ \mathsf{j}^{\rightarrow} = \alpha\ \}$
$\mathsf{k.right} := \mathsf{j}\,;$
$\{\ \mathsf{lr\_tree}(\mathsf{i}) \circledcirc_{\mathsf{LR}} (\mathsf{u\_cell}(\mathsf{j}) \circledcirc_{\mathsf{LR}} \mathsf{u\_cell}(\mathsf{k}, \mathsf{j.left})) \ \wedge\ \mathsf{u\_cell}(\mathsf{k}) \circledcirc_{\mathsf{right}} \mathsf{thread\_list}(\mathsf{j}, \mathsf{k.right})$
$\quad \wedge\ \mathsf{k} \bullet \mathsf{j}^{\rightarrow} = \mathsf{i}^{\rightsquigarrow} \ \wedge\ \mathsf{k} \bullet \mathsf{j}^{\rightarrow} = \mathsf{k} \bullet \alpha\ \}$
$\mathsf{mark}(\mathsf{k})\,;$
$\{\ \mathsf{lr\_tree}(\mathsf{i}) \circledcirc_{\mathsf{LR}} (\mathsf{u\_cell}(\mathsf{j}) \circledcirc_{\mathsf{LR}} \mathsf{m\_cell}(\mathsf{k}, \mathsf{j.left}))$
$\quad \wedge\ \mathsf{m\_cell}(\mathsf{k}) \circledcirc_{\mathsf{thread}} \mathsf{thread\_list}(\mathsf{j}, \mathsf{k.right}) \ \wedge\ \mathsf{k}^{\rightarrow} = \mathsf{i}^{\rightsquigarrow} \ \wedge\ \mathsf{k}^{\rightarrow} = \mathsf{k} \bullet \alpha\ \}$
$\{\ \mathsf{lr\_tree}(\mathsf{i}) \ \wedge\ \mathsf{thread\_list}(\mathsf{k}) \ \wedge\ \mathsf{k}^{\rightarrow} = \mathsf{i}^{\rightsquigarrow} \ \wedge\ \mathsf{k}^{\rightarrow} = \mathsf{k} \bullet \alpha\ \}$
$\mathsf{j} := \mathsf{k}\,;$
$\{\ \mathsf{lr\_tree}(\mathsf{i}) \ \wedge\ \mathsf{thread\_list}(\mathsf{j}) \ \wedge\ \mathsf{j}^{\rightarrow} = \mathsf{i}^{\rightsquigarrow} \ \wedge\ \mathsf{j}^{\rightarrow} = \mathsf{k} \bullet \alpha\ \}$

Figure 5.13: Verification of adding a new node to a threaded tree.

and

$$[\![\, \mathsf{i}^{\rightarrow} \,]\!]_{(s,a)} =_{df} li_{a_{\mathsf{next}}}(s(\mathsf{j}))\,, \qquad [\![\, {}^{\leftarrow}\mathsf{j} \,]\!]_{(s,a)} =_{df} li_{a_{\mathsf{prev}}}(s(\mathsf{j}))\,.$$

We conclude this section by a discussion on related work. There exist several approaches that extend separation logic by additional constructs to include sharing or restrict outgoing pointers of disjoint heaps to a single direction. Wang et al. [WBO08] defined an extension called *Confined Separation Logic* and provided a relational model for it. They defined various operators to assert, e.g., that all outgoing references of a heap h_1 point to another disjoint one h_2 or all outgoing references of h_1 either point to themselves or to h_2.

Our approach is more general due to its algebraicity and hence also able to express the mentioned operations. It is intended as a general foundation for defining further operations and predicates for reasoning about linked object structures.

Another calculus that follows a similar intention as our approach is given in [CS10]. Generally, there heaps are viewed as labelled object graphs. Starting from an abstract foundation the authors define a decidable logic, e.g. for lists, with domain-specific predicates and operations suitable for automated reasoning.

By contrast, our approach enables abstract derivations in a largely first-order algebraic approach, i.e., pointer Kleene algebra [Ehm03]. The given simple (in)equational

laws allow a direct usage of automated theorem proving systems as PROVER9 [McC05] or any other systems through the TPTP LIBRARY [SS98] at the level of the underlying resource algebra [HS08]. This supports and helpfully guides the development of domain specific predicates and operations. The assertions we have presented are simple and still suitable for expressing shapes of linked structures without the need of any arithmetic as in [CS10]. Part of such assertions can be automatically verified using SMALLFOOT [BCO05].

A novel approach to sharing in data structures can be found in [HV13]. This approach can be directly used with arbitrary separation logics and introduces, differing from our approach, an operation called overlapping conjunction. This operator in contrast to the separating conjunction allows unspecified overlapping of the resources characterised by predicates. It enables impressive reasoning about sharing in combination with the separating implication. However, the formulas involved unfortunately become very complex and difficult to understand. We hope that the approach of the present thesis can also capture complex examples like the garbage collecting algorithm given in [HV13] with easier and more concise formulas.

Chapter 6

Conclusion

A variety of algebraic calculi for separation logic has been presented. They range from an algebra for the extended assertion language of separation logic to a relational approach that allowed formulations of the modular behaviour of the involved commands. By this we further developed general and simple approaches to prove the frame and concurrency rules. In particular, we also established relationships to other similar sequential and concurrent approaches. In this chapter we summarise the results and give some open questions for future research.

6.1 Summary

We have developed an abstraction and algebraic calculus for separation logic assertions. The considered denotational model for the abstractions is based on simple sets of states and allowed a further abstraction to the structure of a quantale. By this, pointfree characterisations of assertion classes with particular behaviour have been developed that yield simple proofs of central properties used in concrete verification tasks. This captures, in particular, pure, intuitionistic, precise and supported assertions.

A further relation-based algebraic calculus for commands was introduced that allowed by simple embedding of the algebraic treatment of assertions a reuse of previously gained results. The calculus is enriched by an extra operation on arbitrary relations to model the separating conjunction of separation logic. The definitions allowed again

pointfree versions of central properties, i.e., safety monotonicity and the frame property, that are used for a concise soundness proof of the frame rule. This development also entailed formulations for the preservation of resources as variables, which is handled in separation logic by meta variable side conditions. In addition to this we have presented some further applications with relationships to concurrency and another approach that also copes well with framing called the dynamic frames theory. More concretely, formulations in the context of concurrent separation logic and concurrent Kleene algebras have been derived.

Finally, as the last contribution of this work we developed from the formal and algebraic foundation a transitive separation logic that introduced operations more suitable for reasoning about reachability within linked object structures. For this we enriched the underlying resource algebra in terms of a modal Kleene algebra or more concretely a modal semiring that enabled an abstract characterisation of linked structures. By this we were able to define more specialised operators, in particular for the complete exclusion of sharing within such structures and further restricting the involved links in one direction. Moreover, we presented the treatment in action by a standard example on lists and also more complex examples on trees and overlaid data structures like threaded trees which involves lists and trees.

6.2 Future Work

We presented in this thesis several algebraic calculi and approaches for an abstract and general treatment of behaviours and effects of separation logics. In particular, the developed formalisations yield simple frameworks for the derivation of non-trivial properties and operations that allowed a more adequate and facilitated description of data representations and the structure of resources. However, there are still limitations and open problems that deserve consideration.

As mentioned at the end of Section 3.2.5 the precising operation for supported assertions can be lifted to the abstract setting of quantales. A lot of properties can be proved from the given abstraction in a simple and concise fashion. Unfortunately, it was required to adjust the algebra by axioms for the derivation of some consequences with the given characterisation. This poses the question whether the provided formalisations are in this form adequate enough or if there exist another characterisation and axioms that allow a fully algebraic treatment of that assertion class. A starting point for these investigations can be [BV14]. In that work an extension of the related algebraic approach of Boolean BI algebras is introduced that further allows the inclusion of basic and frequently used properties of separation or resource algebras that was not possible with the former approach. As further investigations on this one can consider algebraic treatments of non-discussed assertion classes as e.g., *strictly exact*

assertions [Rey08].

In Section 4.4.2 we discussed the generalised version of separating conjunction to characterise concurrent non-interfering programs. The relational definition of separation includes the extension to pairs of relations by Cartesian products that allows an independent treatment of programs on individual parts of states. We suppose that an extension of the definition of separation at this point may enable a more precise and adequate treatment of races like in [COY07] within the relational notation. One would require a definition on pairs of commands that also includes possible interactions of the parallel executed programs. In particular, this might entail an equational relationship in $[\![C \parallel D]\!]_\Gamma \subseteq [\![C]\!]_\Gamma * [\![D]\!]_\Gamma$ and thus possibly give a relational model of concurrent Kleene algebras by verifying soundness of the exchange law (cf. Section 4.4.3).

Further treatments and extended approaches for the dynamic frames theory need to be compared to our relational abstractions. This yields in particular more applications for the theory and again abstractions that conversely help to guide the development of further separation logics. We summarise a few approaches that fit into this research as, e.g., [DYDG$^+$10] where the notion of *concurrent abstract predicates* is introduced that includes abstractions for reasoning about shared portions of storage within concurrent contexts. Relationships to this theory would immediately relate the dynamic frames treatment also to the topic of concurrency. As similar approach is taken by *fictional separation logic* [JB12] that defines on top of separation logic a meta-theory that allows modular reasoning about data types that include sharing. Moreover it would be interesting to investigate the relationships to our extension to transitive separation logic in Chapter 5. A further research direction that deserves attention concentrates on the relationships of implicit dynamic frames, i.e., a variation to the original theory that captures aspects of automation, to separation logic [PS11]. These results might help to design and develop further investigations on relational formulations for the theory of Section 4.5.

Finally, a general intention of the present work was to relate the earlier approach of pointer Kleene algebra with concepts of separation logic which resulted in transitive separation logic. In particular, we obtained more suitable operators for reasoning about structural data structures and their sharing relationships. As future work, it will be interesting to explore more complex object structures and verify challenging garbage collecting algorithms. For this standard examples in the literature are the *Schorr-Waite graph marking* algorithm [SW67] and further adaptations for the treatment of concurrent garbage collection algorithms. A derivational approach to this is given in [PPS10] while first formulations towards this topic within pointer Kleene algebra can be found in [Dan12].

Appendix A

Deferred Proofs and Properties

A.1 Deferred Proofs

Proof of Lemma 3.1.7. By Lemma 3.1.4 we have $[\![p \mathrel{-\!\!*} q]\!] = [\![p]\!] \backslash [\![q]\!]$. Now it is easy to see that

$$
\begin{array}{ll}
 & s,h \models p \mathrel{-\!\!\circledast} q \\
\Leftrightarrow & \quad \{\![\text{ definition of } \mathrel{-\!\!\circledast}]\!\} \\
 & s,h \models \neg(p \mathrel{-\!\!*} (\neg q)) \\
\Leftrightarrow & \quad \{\![\text{ definition of } \mathrel{-\!\!*}]\!\} \\
 & \neg(\forall h' : ((dom(h') \cap dom(h) = \emptyset,\ s,h' \models p) \ \Rightarrow\ s, h' \cup h \models \neg q)) \\
\Leftrightarrow & \quad \{\![\text{ logic: } \neg \text{ over } \forall]\!\} \\
 & \exists\, h' : \neg((dom(h') \cap dom(h) = \emptyset,\ s,h' \models p) \ \Rightarrow\ s, h' \cup h \models \neg q) \\
\Leftrightarrow & \quad \{\![\text{ logic: } \neg(A \ \Rightarrow\ B) \ \Leftrightarrow\ (A \wedge \neg B)]\!\} \\
 & \exists\, h' : dom(h') \cap dom(h) = \emptyset \wedge s,h' \models p \wedge s, h' \cup h \not\models \neg q \\
\Leftrightarrow & \quad \{\![\text{ logic }]\!\} \\
 & \exists\, h' : dom(h') \cap dom(h) = \emptyset \wedge s,h' \models p \wedge s, h' \cup h \models q \\
\Leftrightarrow & \quad \{\![\text{ setting for } (\Rightarrow)\ \hat{h} =_{df} h' \cup h \text{ and for } (\Leftarrow)\ h' =_{df} \hat{h} - h]\!\} \\
 & \exists\, \hat{h} : h \subseteq \hat{h} \wedge s, \hat{h} - h \models p \wedge s, \hat{h} \models q\,.
\end{array}
$$

□

Proof of Lemma 3.2.7.

(a) The claim follows immediately from Lemma 3.2.10, (exc) and part (b).

(b) We first show that $a = (a \sqcap 1) \cdot \top$ follows from both inequations. By neutrality of $\top$ and 1 w.r.t. $\sqcap$ and $\cdot$ resp., assumption and isotony, we get

$$a = a \sqcap \top = a \sqcap 1 \cdot \top \leq (a \sqcap 1) \cdot (a \sqcap \top) \leq (a \sqcap 1) \cdot \top .$$

Moreover, by isotony and assumption $(a \sqcap 1) \cdot \top \leq a \cdot \top \leq a$. Next we show that $a = (a \sqcap 1) \cdot \top$ implies $a \cdot \top \leq a$ and $\overline{a} \cdot \top \leq \overline{a}$ which, by part (a), implies the claim. For the first inequation we calculate by assumption, $\top \cdot \top = \top$ and the assumption again: $a \cdot \top = (a \sqcap 1) \cdot \top \cdot \top = (a \sqcap 1) \cdot \top = a$. For the second inequation, we note that in a Boolean quantale the law $\overline{t \cdot \top} = (\overline{t} \sqcap 1) \cdot \top$ holds for all subidentities t ($t \leq 1$) (e.g. [DM01a]). From this we get

$$\overline{a} \cdot \top = \overline{(a \sqcap 1) \cdot \top} \cdot \top = (\overline{a} \sqcap 1) \cdot \top \cdot \top = (\overline{a} \sqcap 1) \cdot \top = \overline{(a \sqcap 1) \cdot \top} = \overline{a} .$$

(c) We show the equivalence of $(a \sqcap b) \cdot c = a \sqcap b \cdot c$ with part (b). First we prove the $\Leftarrow$-direction and split the proof showing each inequation separately. Assuming the distributivity property of part (b) we show $\geq$ by

$$a \sqcap b \cdot c \leq (a \sqcap b) \cdot (a \sqcap c) \leq (a \sqcap b) \cdot c .$$

For the $\leq$-direction we know $(a \sqcap b) \cdot c \leq a \cdot c \leq a \cdot \top \leq a$ which follows from isotony and the assumption $a \cdot \top \leq a$. Now, with $(a \sqcap b) \cdot c \leq b \cdot c$ we can immediately conclude $(a \sqcap b) \cdot c \leq a \sqcap b \cdot c$.

Next we give a proof for the $\Rightarrow$-direction and assume $(a \sqcap b) \cdot c = a \sqcap b \cdot c$ holds. First, $a \cdot \top = (a \sqcap a) \cdot \top = a \sqcap a \cdot \top \leq a$. Furthermore, we calculate

$$a \sqcap b \cdot c = a \sqcap a \sqcap b \cdot c = a \sqcap (a \sqcap b) \cdot c = a \sqcap c \cdot (a \sqcap b) = (a \sqcap b) \cdot (a \sqcap c)$$

which follows from idempotence of $\sqcap$, assumption, commutativity of $\cdot$ and again assumption and commutativity. □

Proof of Lemma 3.2.10. ($\Leftarrow$): Using Equation (Ded), isotony and the assumption, we get

$$a \sqcap b \cdot c \leq (a \lfloor c \sqcap b) \cdot c \leq (a \lfloor \top \sqcap b) \cdot c \leq (a \sqcap b) \cdot c$$

and the symmetric formula $a \sqcap b \cdot c \leq b \cdot (a \sqcap c)$ for arbitrary elements a, b, c. Now, the claim follows using idempotence of $\sqcap$, isotony and the above derived inequations in

$$a \sqcap b \cdot c = a \sqcap a \sqcap b \cdot c \leq a \sqcap (a \sqcap b) \cdot c \leq (a \sqcap b) \cdot (a \sqcap c) .$$

($\Rightarrow$): By setting $b = \overline{a}$ and $c = \top$ in the distributivity law, we obtain $a \sqcap \overline{a} \cdot \top \leq (a \sqcap \overline{a}) \cdot (a \sqcap \top) = 0 \cdot a = 0$ and hence, by (shu) $\overline{a} \cdot \top \leq \overline{a}$. This is equivalent to $a \lfloor \top \leq a$ using (exc). □

Proof of Lemma 3.2.26.

(a) Since $[\![(s,h)]\!]$ is a singleton set, this is obvious.

(b) By definition of $*$, Part (a), $h \subseteq h' \wedge \hat{h} \subseteq h'$ implies $h' - \hat{h} = h' - h \Leftrightarrow \hat{h} = h$, and set theory:

$$\begin{aligned}
& s, h' \models p * (s, h' - h) \\
\Leftrightarrow\ & \exists\, \hat{h} : \hat{h} \subseteq h' \wedge s, \hat{h} \models p \wedge s, h' - \hat{h} \models (s, h' - h) \\
\Leftrightarrow\ & \exists\, \hat{h} : \hat{h} \subseteq h' \wedge s, \hat{h} \models p \wedge h' - \hat{h} = h' - h \\
\Leftrightarrow\ & \exists\, \hat{h} : \hat{h} \subseteq h' \wedge s, \hat{h} \models p \wedge \hat{h} = h \\
\Leftrightarrow\ & s, h \models p .
\end{aligned}$$

(c) By definition of $*$, $s, h' - \hat{h} \models \mathsf{true}$ is true, Part (1), and set theory:

$$\begin{aligned}
& s, h' \models (s, h) * \mathsf{true} \\
\Leftrightarrow\ & \exists\, \hat{h} : \hat{h} \subseteq h' \wedge s, \hat{h} \models (s, h) \wedge s, h' - \hat{h} \models \mathsf{true} \\
\Leftrightarrow\ & \exists\, \hat{h} : \hat{h} \subseteq h' \wedge s, \hat{h} \models (s, h) \\
\Leftrightarrow\ & \exists\, \hat{h} : \hat{h} \subseteq h' \wedge \hat{h} = h \\
\Leftrightarrow\ & h \subseteq h' .
\end{aligned}$$

□

Proof of Lemma 4.3.4. For the first claim we calculate as follows.

$$\begin{aligned}
& (\sigma_1, \sigma_2) \ \# \ (\tau_1, \tau_2) \\
\Leftrightarrow\ & \quad \{\!\!\{ \text{ Definition 4.3.3 } \}\!\!\} \\
& \sigma_1 \# \sigma_2 \wedge \sigma_1 = \tau_1 \wedge \sigma_2 = \tau_2 \\
\Leftrightarrow\ & \quad \{\!\!\{ \text{ logic } \}\!\!\} \\
& \sigma_1 \# \sigma_2 \wedge \tau_1 \# \tau_2 \wedge \sigma_1 = \tau_1 \wedge \sigma_2 = \tau_2
\end{aligned}$$

$$
\begin{array}{ll}
\Leftrightarrow & \{\![\text{ Definition 3.3.1 }]\!\} \\
& \sigma_1 \# \sigma_2 \wedge \exists \sigma : \sigma_1 \bullet \sigma_2 = \sigma \wedge \tau_1 \# \tau_2 \wedge \exists \tau : \tau_1 \bullet \tau_2 = \tau \wedge \sigma_1 = \tau_1 \wedge \sigma_2 = \tau_2 \\
\Leftrightarrow & \{\![\ \sigma_1 = \tau_1 \wedge \sigma_2 = \tau_2 \text{ implies } \sigma = \tau \ ]\!\} \\
& \exists \sigma : \ \sigma_1 \# \sigma_2 \wedge \sigma_1 \bullet \sigma_2 = \sigma \wedge \tau_1 \bullet \tau_2 = \sigma \wedge \tau_1 \# \tau_2 \wedge \sigma_1 = \tau_1 \wedge \sigma_2 = \tau_2 \\
\Leftrightarrow & \{\![\text{ logic }]\!\} \\
& \exists \sigma : \ (\sigma_1, \sigma_2) \ \rhd \sigma \wedge \sigma \lhd \ (\tau_1, \tau_2) \wedge \sigma_1 = \tau_1 \wedge \sigma_2 = \tau_2 \\
\Leftrightarrow & \{\![\text{ definition of } ; \ ]\!\} \\
& (\sigma_1, \sigma_2) \ (\rhd \, ; \lhd) \ (\tau_1, \tau_2) \wedge \sigma_1 = \tau_1 \wedge \sigma_2 = \tau_2 \\
\Leftrightarrow & \{\![\text{ definition of } \mathsf{id} \ ]\!\} \\
& (\sigma_1, \sigma_2) \ (\rhd \, ; \lhd \cap \ \mathsf{id}) \ (\tau_1, \tau_2)
\end{array}
$$

The remaining claims are immediate from the definitions. □

Proof of Lemma 4.3.7. For Part (a) we have by (abortext) for arbitrary states σ, τ:

$$\neg(\sigma \bullet \tau = \sigma_\perp) \ \Leftrightarrow \ \neg(\sigma = \sigma_\perp) \wedge \neg(\tau = \sigma_\perp) \, .$$

Hence, for arbitrary states σ, τ, ρ we reason as follows

$$
\begin{array}{ll}
\rho \ (\neg\bot \, ; \lhd) \ (\sigma, \tau) & \Leftrightarrow \neg(\rho = \sigma_\perp) \wedge \rho \ \lhd \ (\sigma, \tau) \\
& \Leftrightarrow \neg(\rho = \sigma_\perp) \wedge \rho = \sigma \bullet \tau \wedge \sigma \ \# \ \tau \\
& \Leftrightarrow \neg(\sigma \bullet \tau = \sigma_\perp) \wedge \rho = \sigma \bullet \tau \wedge \sigma \ \# \ \tau \\
& \Leftrightarrow \neg(\sigma = \sigma_\perp) \wedge \neg(\tau = \sigma_\perp) \wedge \rho = \sigma \bullet \tau \wedge \sigma \ \# \ \tau \\
& \Leftrightarrow \rho \ \lhd \, ; (\neg\bot \times \neg\bot) \ (\sigma, \tau)
\end{array}
$$

by the definitions and the above equivalence. Part (b) and (c) immediately follows from (abortext). □

Proof of Lemma 4.3.10. For arbitrary σ we have

$$
\begin{array}{ll}
& \sigma \ \lceil(R * S) \ \sigma \\
\Leftrightarrow & \{\![\text{ Definition 4.3.5 and definition of domain }]\!\} \\
& \exists \sigma_1, \sigma_2, \tau_1, \tau_2 \, . \, \sigma_1 \# \sigma_2 \wedge \sigma = \sigma_1 \bullet \sigma_2 \wedge \tau_1 \# \tau_2 \wedge \sigma_1 \ R \ \tau_1 \wedge \sigma_2 \ S \ \tau_2 \\
\Rightarrow & \{\![\text{ omitting conjunct } \tau_1 \# \tau_2 \text{ and shifting quantification over } \tau_1, \tau_2 \ ]\!\} \\
& \exists \sigma_1, \sigma_2 \, . \, \sigma_1 \# \sigma_2 \wedge \sigma = \sigma_1 \bullet \sigma_2 \wedge \exists \tau_1, \tau_2 \, . \, \sigma_1 \ R \ \tau_1 \wedge \sigma_2 \ S \ \tau_2 \\
\Leftrightarrow & \{\![\text{ definition of domain }]\!\} \\
& \exists \sigma_1, \sigma_2 \, . \, \sigma_1 \# \sigma_2 \wedge \sigma = \sigma_1 \bullet \sigma_2 \wedge \sigma_1 \ \lceil R \ \sigma_1 \wedge \sigma_2 \ \lceil S \ \sigma_2 \\
\Leftrightarrow & \{\![\text{ Definition 4.3.5 }]\!\} \\
& \sigma \ (\lceil R * \lceil S) \ \sigma \, .
\end{array}
$$

□

Proof of Lemma 4.3.18. Assume a compensator K. We first show the case for total correctness. Closure under union is straightforward from additivity of domain and distributivity of ; and $\times$ over $\cup$. For closure under composition we begin with an auxiliary result and show

$$\lceil(C\,;D)\,;C \;\subseteq\; C\,;\lceil D\,. \tag{A.1}$$

First, $\lceil(C;D) = \lceil(C;\lceil D)$. By distributivity, $C = C;\lceil D \cup C;\neg\lceil D$ and using $R = \lceil R;R$ for any relation R we can infer $\lceil(C\,;D)\,;C = \lceil(C\,;D)\,;C\,;\lceil D \;\subseteq\; C\,;\lceil D$.

Assume now that C and D have the frame property. By $\lceil(C\,;D) \;\subseteq\; \lceil C$, neutrality and exchange ($\times$/;), C has the frame property, neutrality and exchange ($\times$/;), Equation (A.1), neutrality and exchange ($\times$/;), D has the frame property, exchange ($\times$/;), and by definition of compensators:

$$\begin{array}{ll}
 & (\lceil(C\,;D)\times I)\,;\rhd\,;\,C\,;D\\
\subseteq & (\lceil(C\,;D)\,;\lceil C\;\times I)\,;\rhd\,;\,C\,;D\\
= & (\lceil(C\,;D)\times I)\,;(\lceil C\times I)\,;\rhd\,;\,C\,;D\\
\subseteq & (\lceil(C\,;D)\times I)\,;(C\times K)\,;\rhd\,;D\\
= & (\lceil(C\,;D)\,;C\;\times K)\,;\rhd\,;D\\
\subseteq & (C\,;\lceil D\;\times K)\,;\rhd\,;D\\
= & (C\times K)\,;(\lceil D\times I)\,;\rhd\,;D\\
\subseteq & (C\times K)\,;(D\times K)\,;\rhd\\
= & (C\,;D\;\times\;K\,;K)\,;\rhd\\
\subseteq & (C\,;D\times K)\,;\rhd\,.
\end{array}$$

Next we show the case for partial correctness. Closure under union follows from Lemma 4.2.5, distributivity and isotony. For closure under composition we first state and prove the auxiliary result

$$\mathsf{safe}\,(C\,;D)\,;C \;\subseteq\; C\,;\mathsf{safe}\,(D)\,. \tag{A.2}$$

First, note that $\mathsf{safe}\,(C\,;D) = \neg\lceil(C\,;\lceil(D\,;\bot))$. and by distributivity, $C = C\,;\lceil(D\,;\bot) \cup C\,;\neg\lceil(D\,;\bot)$. Using $R = \lceil R\,;R$ we immediately infer

$$\mathsf{safe}\,(C\,;D)\,;C = \mathsf{safe}\,(C\,;D)\,;C\,;\neg\lceil(D\,;\bot) \;\subseteq\; C\,;\mathsf{safe}\,(D)\,.$$

Note that the auxiliary result (A.2) coincides structurally with (A.1) substituting $\mathsf{safe}\,(\_)$ with $\lceil\_$. Now by the use of the assumption that D respects $\bot$, Lemma 4.2.5 and $\neg\bot$ is idempotent w.r.t. ;, we can largely reuse the above proof for the case of

partial correctness by replacing all occurrences of $\lceil\_$ with $\mathsf{safe}\,(\_)$. It remains the application of Definition 4.3.13(d) after the third $\subseteq$ to get formula into the form for applying the frame property for D. □

Proof of Equation (4.4) **being equivalent to Definition 4.3.20.** First, assume $\# ; (C \times r ; K) ; \# \subseteq \# ; (\top \times \top ; r) ; \#$. By Lemma 4.3.4, the assumption, exchange ($\times$/;), tests commute and are idempotent, we calculate

$$\begin{aligned} \lhd ; (C \times r ; K) ; \# &= \lhd ; \# ; (C \times r ; K) ; \# \\ &\subseteq \lhd ; (T \times T ; r) ; \# \\ &= \lhd ; (T \times T) ; \# ; (I \times r) ; \# \\ &= \top ; \lhd ; (I \times r) . \end{aligned}$$

Now, assume $\lhd ; (C \times r ; K) ; \# \subseteq \top ; \lhd ; (I \times r)$. By the argumentation above the right-hand side equals $\lhd ; (\top \times \top ; r) ; \#$. Hence, we can calculate by Lemma 4.3.4, the assumption, isotony, exchange ($\times$/;) and $\top ; \top = \top$

$$\begin{aligned} \# ; (C \times r ; K) ; \# &\subseteq \rhd ; \lhd ; (C \times r ; K) ; \# \\ &\subseteq \rhd ; \lhd ; (\top \times \top ; r) ; \# \\ &\subseteq \# ; (\top \times \top) ; (\top \times \top ; r) ; \# \\ &= \# ; (\top \times \top ; r) ; \# . \end{aligned}$$

□

Proof of Lemma 4.3.23. Assume a compensator K, a test p and that C preserves a test r. Then

$$\lhd ; (p ; C \times r ; K) ; \# \subseteq \lhd ; (C \times r ; K) ; \# \subseteq \top ; \lhd ; (I \times r) .$$

Moreover closure under union is straightforward from distributivity of ; and $\times$ over $\cup$. Now assume further that D preserves r and

$$\# ; (C ; D \times K) ; \# \subseteq \# ; (C \times K) ; \# ; (D \times K) ; \# .$$

By neutrality of I and exchange ($\times$/;), by Lemma (4.3.4), $\#$ and $I \times r$ are tests and commute, by assumption, isotony, exchange ($\times$/;) and neutrality of I, C preserves r, again exchange ($\times$/;) and neutrality of I, D preserves r, and definition of $\top$:

$$\begin{aligned} & \lhd ; (C ; D \times r ; K) ; \# \\ = \; & \lhd ; (I \times r) ; (C ; D \times K) ; \# \\ = \; & \lhd ; \# ; (I \times r) ; (C ; D \times K) ; \# \end{aligned}$$

$$
\begin{array}{ll}
= & \lhd\,;(I\times r)\,;\#\,;(C\,;D\,\times K)\,;\# \\
\subseteq & \lhd\,;(I\times r)\,;\#\,;(C\times K)\,;\#\,;(D\times K)\,;\# \\
\subseteq & \lhd\,;(I\times r)\,;(C\times K)\,;\#\,;(D\times K)\,;\# \\
= & \lhd\,;(C\times\, r\,;K)\,;\#\,;(D\times K)\,;\# \\
\subseteq & \top\,;\lhd\,;(I\times r)\,;(D\times K)\,;\# \\
= & \top\,;\lhd\,;(D\times\, r\,;K)\,;\# \\
\subseteq & \top\,;\top\,;\lhd\,;(I\times r) \\
\subseteq & \top\,;\lhd\,;(I\times r)\,.
\end{array}
$$

□

Proof of Lemma 4.3.26. We first show the second equivalence. By by $I \subseteq \top$, isotony and neutrality, again isotony and composing $\top$ to both sides, and $\top\,;\top \subseteq \top$:

$$
\begin{array}{ll}
 & \top\,;p\,;C\subseteq\top\,;q \\
\Rightarrow & p\,;C\subseteq\top\,;q \\
\Rightarrow & \top\,;p\,;C\subseteq\top\,;\top\,;q \\
\Rightarrow & \top\,;p\,;C\subseteq\top\,;q\,.
\end{array}
$$

Next we tackle the first equivalence. For the $\Rightarrow$-direction we assume $p\,;C \subseteq C\,;q$ and infer by isotony that $\top\,;p\,;C \subseteq \top\,;C\,;q \subseteq \top\,;\top\,;q \subseteq \top\,;q$.

For the converse we have since $p \subseteq I$ that $p\,;C \subseteq C$. Moreover, by neutrality, isotony and the assumption, we calculate $p\,;C \subseteq \top\,;p\,;C \subseteq \top\,;q$. Thus, $p\,;C \subseteq C \cap \top\,;q$. Finally, since relations more abstractly form test semirings we can infer by a standard result on that structures (e.g. [Möl07]) that the right-hand side equals $C\,;q$. □

Proof of Lemma 4.3.27.

(a) By $p \subseteq \lceil C$ and $\cap$ coincides with ; on tests, definition of $*$ (Definition 4.3.5), exchange ($\times$/;), C has the generalised frame property, exchange ($\times$/;), and definition of $*$ (Definition 4.3.5) again:

$$
\begin{array}{ll}
 & (p*r)\,;C \\
\subseteq & ((p\,;\lceil C)*r)\,;C \\
= & \lhd\,;\,(p\,;\lceil C\,\times r)\,;\rhd\,;C \\
= & \lhd\,;\,(p\times r)\,;(\lceil C\times I)\,;\rhd\,;C \\
\subseteq & \lhd\,;\,(p\times r)\,;(C\times K)\,;\rhd \\
= & \lhd\,;\,(p\,;C)\times(r\,;K)\,;\rhd \\
= & (p\,;C)*(r\,;K)\,.
\end{array}
$$

The other case follows from substituting $\lceil C$ with $\mathsf{safe}\,(C)$, r with $r\,;\neg\bot$ and using exchange ($\times$/;) for the third step s.t. $\mathsf{safe}\,(C)\times\neg\bot$ is composed instead of $\lceil C\times I$.

(b) By definition of $*$ (Definition 4.3.5), neutrality of I and exchange $(\times/;)$, Lemma 4.3.4, $(q \times I)$ and $\#$ commute, C preserves r, exchange $(\times/;)$ and neutrality of I, and definition of $*$ (Definition 4.3.5):

$$\begin{array}{ll} & (C\,;q) * (r\,;K) \\ = & \lhd\,;\,(C\,;q) \times (r\,;K)\,;\rhd \\ = & \lhd\,;\,(C \times (r\,;K))\,;(q \times I)\,;\rhd \\ = & \lhd\,;\,(C \times (r\,;K))\,;(q \times I)\,;\#\,;\rhd \\ = & \lhd\,;\,(C \times (r\,;K))\,;\#\,;(q \times I)\,;\rhd \\ \subseteq & \top\,;\lhd\,;(I \times r)\,;(q \times I)\,;\rhd \\ = & \top\,;\lhd\,;(q \times r)\,;\rhd \\ = & \top\,;(q * r)\,. \end{array}$$

□

Proof of Lemma 4.4.15. The $\subseteq$-direction follows from isotonicity. For the other direction we calculate: By ; coincides on tests with $\cap$, definition of $*$, neutrality, $(\times/;)$, p satisfies (4.13), $(\times/;)$, neutrality, definition of $*$, and ; on tests equals $\cap$,

$$\begin{array}{ll} & p * q \cap p * r \\ = & (p * q)\,;(p * r) \\ = & \lhd\,;(I\,;p \times q\,;I)\rhd\,;\lhd\,;(p\,;I \times I\,;r)\,;\rhd \\ = & \lhd\,;(I \times q)\,;(p \times I)\rhd\,;\lhd\,;(p \times I)\,;(I \times r)\,;\rhd \\ \subseteq & \lhd\,;(I \times q)\,;(p \times I)\,;(I \times r)\,;\rhd \\ = & \lhd\,;(p \times (q\,;r))\,;\rhd \\ = & p * (q \cap r)\,. \end{array}$$

□

For a proof Lemma 4.4.22 we require an auxiliary result.

Lemma A.1.1 *If relations P, Q are forward compatible then* $(P\,;\top) * (Q\,;\top) = (P * Q)\,;\top$.

Proof. We calculate By definition of $*$, Lemma 4.3.4, Equation $(\times/;)$, P, Q forward compatible, Lemma 4.3.25, and definition of $*$:

$$\begin{array}{ll} & (P\,;\top) * (Q\,;\top) \\ = & \lhd\,;(P\,;\top \times Q\,;\top)\,;\rhd \\ = & \lhd\,;\#\,;(P \times Q)\,;(\top \times \top)\,;\rhd \\ \subseteq & \lhd\,;(P \times Q)\,;\#\,;(\top \times \top)\,;\rhd \\ = & \lhd\,;(P \times Q)\,;\rhd\,;\top \\ = & (P * Q)\,;U\,. \end{array}$$

The reverse inequation follows similarly from $\# \subseteq \mathsf{id}$. □

Finally we are able to prove Lemma 4.4.22.

Proof of Lemma 4.4.22. First note that $\ulcorner P = P\,;\top \cap I$. The same holds for Q. By this we calculate

$$\ulcorner P * \ulcorner Q = (P\,;\top \cap I) * (Q\,;\top \cap I) \subseteq (P\,;\top) * (Q\,;\top) = (P * Q)\,;\top\,.$$

Moreover $\ulcorner P * \ulcorner Q \subseteq I$, since both are tests. Hence we can conclude $\ulcorner P * \ulcorner Q \subseteq (P * Q)\,;U \cap I = \ulcorner(P * Q)$. The reverse inclusion was shown in Lemma 4.3.10. □

The combinability check $\#$ is a test on pairs of relations. Hence, it induces some useful closure properties that we list in the following.

Corollary A.1.2 *If P, Q are forward compatible and $R \subseteq P$ then also R, Q are forward compatible. This result also holds for backward compatibility, hence compatibility is downward closed, too.*

Proof. We show the following more general result: Let C, D, E be relations on pairs of states such that C is a test. If C is an invariant of D, i.e., $C\,;D \subseteq D\,;C$, and $E \subseteq D$ then C is also an invariant of E. For this we calculate

$$\begin{aligned} C\,;E = C\,;(D \cap E) = C\,;D \cap C\,;E \subseteq D\,;C \cap C\,;E = \\ D \cap C\,;E\,;C \subseteq C\,;E\,;C \subseteq E\,;C\,. \end{aligned}$$

The fourth step follows, since C is a test. A proof can, e.g., be found in [Möl07]. Now the main claim follows by setting $C = \#$, $D = P \times Q$ and $E = R \times Q$. □

Note that this proof extends to arbitrary test semirings.

Corollary A.1.3 *If P is forward/backward compatible with Q and R then it is also forward/backward compatible with $Q \cup R$.*

Proof. We show the case of forward compatibility:

$$\begin{aligned} \#\,;(P \times (Q \cup R)) = \#\,;((P \times Q) \cup (P \times R)) = \#\,;(P \times Q) \cup \#\,;(P \times R) \subseteq \\ (P \times Q)\,;\# \cup (P \times R)\,;\# = ((P \times Q) \cup (P \times R))\,;\# = (P \times (Q \cup R))\,;\#\,. \end{aligned}$$

□

Corollary A.1.4

a) Let P, Q and R, S be forward compatible. Then also $P\,;R$ and $Q\,;S$ are forward compatible. Again the same holds for backward compatibility.

b) If P and Q are forward/backward compatible then so are P^n and Q^n for all $n \in \mathbb{N}$, where the n-th power of a command means n-fold sequential composition of the command with itself. Hence also $P^n * Q^n \subseteq (P * Q)^n$.

Proof.

a) $\#\,;(P\,;R \times Q\,;S) = \#\,;(P \times Q)\,;(R \times S) \subseteq (P \times Q)\,;\#\,;(R \times S) \subseteq (P \times Q)\,;(R \times S)\,;\# = (P\,;R \times Q\,;S)\,;\#$.

b) Straightforward induction on n using Part a) and the reverse exchange law. □

Proof of Lemma 4.4.31. We set $Q_i = P_i\,;r_i$ and $p_i = \ulcorner Q_i$ in Theorem 4.4.29. This validates the premise of the rule, since $\{\ulcorner Q_i\}\,Q_i\,\{r_i\} \;\Leftrightarrow\; \ulcorner Q_i\,;Q_i \subseteq Q_i\,;r_i \wedge \ulcorner Q_i \subseteq \ulcorner Q_i \;\Leftrightarrow\; Q_i \subseteq Q_i\,;r_i$ and $Q_i\,;r_i = (P_i\,;r_i)\,;r_i = P_i\,;r_i = Q_i$. Hence, by the conclusion of the rule we get

$$(\ulcorner Q_1 * \ulcorner Q_2)\,;(Q_1 * Q_2) \subseteq (Q_1 * Q_2)\,;(r_1 * r_2)\ . \tag{A.3}$$

Next we calculate: By definitions of Q_i , property of domain, by Lemma 4.3.10, by (A.3), definitions of Q_i , and by $r_i \subseteq I$:

$$(P_1\,;r_1) * (P_2\,;r_2) = Q_1 * Q_2 = \ulcorner(Q_1 * Q_2)\,;(Q_1 * Q_2) \subseteq (\ulcorner Q_1 * \ulcorner Q_2)\,;(Q_1 * Q_2) \subseteq (Q_1 * Q_2)\,;(r_1 * r_2) = ((P_1\,;r_1) * (P_2\,;r_2))\,;(r_1 * r_2) \subseteq (P_1 * P_2)\,;(r_1 * r_2)\,.$$

□

Nest, we recapitulate Equation (4.9):

$$\lhd\,;(C \times r)\,;\# \subseteq C\,;\lhd\,;(I \times r)$$

and say that C *s-preserves* a test r iff this condition is satisfied. We list a few useful properties in connection with these notions.

Lemma A.1.5

(a) *I s-preserves I.*

(b) *For any relation C and test r we have $\lhd\,;(C \times r)\,;\# \subseteq (C * I)\,;\lhd\,;(I \times r)$.*

(c) *If C s-preserves a test r then $C * r \subseteq C\,;(I * r)$. In particular, $I * I \subseteq I$. Hence if C s-preserves any test r then $C * I \subseteq C$.*

Proof.

(a) The claim follows immediately by setting $C = I = r$ in Equation (4.9).

(b) We calculate: By Equation $(\times/;)$, neutrality, $I \times r$ and $\#$ are tests and commute, Lemma 4.3.4, definition of $*$,

$$\begin{array}{ll} & \lhd\,;\,(C \times r)\,;\,\# \\ = & \lhd\,;\,(C \times I)\,;\,(I \times r)\,;\,\# \\ = & \lhd\,;\,(C \times I)\,;\,\#\,;\,(I \times r) \\ \subseteq & \lhd\,;\,(C \times I)\,;\,\rhd\,;\,\lhd\,;\,(I \times r) \\ = & (C * I)\,;\,\lhd\,;\,(I \times r)\,. \end{array}$$

(c) The first claim is immediate from the definition of locality by right-composing both sides of the inclusion with $\rhd$, isotony and the definition of $*$. Hence the second claim is trivial by isotony. The third claim follows by setting $r = I$ and using $I * I = I$.

□

We can now give the

Proof of Lemma 4.3.31. The direction $(\Rightarrow)$ is just Lemma A.1.5(c). For $(\Leftarrow)$ we obtain by Lemma A.1.5(b) and the assumption, for arbitrary test r,

$$\lhd\,;\,(C \times r)\,;\,\# \;\subseteq\; (C * I)\,;\,\lhd\,;\,(I \times r) \;\subseteq\; C\,;\,\lhd\,;\,(I \times r)\,.$$

□

Corollary A.1.6 $C * I \subseteq C \;\Leftrightarrow\; \lhd\,;(C \times I)\,;\,\# \subseteq C\,;\lhd$.

Proof. The direction $(\Leftarrow)$ follows from composing both sides with $\rhd$. For the other direction we immediately get by definition and isotony $\lhd\,;(Q \times I)\,;\rhd\,;\lhd \subseteq Q\,;\lhd$, since $C * I \subseteq C$. Now the claim follows from Lemma 4.3.4 using $\# \subseteq \rhd\,;\lhd$. □

Proof of Lemma 4.5.7. By Theorem 4.5.3 and definition of $*$, f is a test, $I = I\,;I$ and Equation $(\times/;)$, Lemma 4.4.23, Lemma 4.3.4, Theorem 4.5.3, and definition of $*$,

$$\begin{array}{ll} \Delta f & = \lhd\,;(f\,;\top \times I)\,;\rhd \\ & = \lhd\,;(f\,;f\,;\top \times I)\,;\rhd \\ & = \lhd\,;(f \times I)\,;(f\,;\top \times I)\,;\rhd \\ & = \lhd\,;(f \times I)\,;\#\,;(f\,;\top \times I)\,;\rhd \\ & \subseteq \lhd\,;(f \times I)\,;\rhd\,;\lhd\,;(f\,;\top \times I)\,;\rhd \\ & = (f * I)\,;\Delta f\,. \end{array}$$

Hence, by (reldom) we have $\lceil(\Delta f) \subseteq f * I$. For the converse we calculate $f * I = \lceil(f * I) \subseteq \lceil((f ; \top) * I) = \lceil(\Delta f)$. Analogous calculations show the result for Ξf. □

Proof of Lemma 4.5.9.

$$
\begin{array}{ll}
& \sigma_1 \ \lceil(\Delta f)\ \sigma_1 \wedge \sigma_1 \bullet \sigma_2 \ \ \Delta f \ \ \sigma' \\
\Leftrightarrow & \quad \{\![\text{ Lemma 4.5.8 and Lemma 4.5.7 }]\!\} \\
& \sigma_1 \ f * I \ \sigma_1 \wedge \sigma_1 \bullet \sigma_2 \ \ \Delta f * I \ \ \sigma' \\
\Leftrightarrow & \quad \{\![\text{ definition of } *]\!\} \\
& \exists \sigma_f, \sigma_I, \tau_1, \tau_2, \tau_1'.\ \ \sigma_1 = \sigma_f \bullet \sigma_I \wedge \sigma_f \in f \wedge \sigma_1 \bullet \sigma_2 = \tau_1 \bullet \tau_2 \\
& \wedge\ \tau_1 \ \Delta f \ \tau_1' \wedge \sigma' = \tau_1' \bullet \tau_2 \\
\Leftrightarrow & \quad \{\![\text{ Theorem 4.5.3 }]\!\} \\
& \exists \sigma_f, \sigma_I, \tau_1, \tau_2, \tau_1', \tau_f, \tau_f', \tau_I.\ \ \sigma_1 = \sigma_f \bullet \sigma_I \wedge \sigma_f \in f \wedge \\
& \sigma_1 \bullet \sigma_2 = \tau_1 \bullet \tau_2 \wedge \ \ \tau_1 = \tau_f \bullet \tau_I \wedge \tau_1' = \tau_f' \bullet \tau_I \wedge \tau_f \in f \wedge \\
& \sigma' = \tau_1' \bullet \tau_2 \\
\Rightarrow & \quad \{\![\text{ Equation (4.13) implies } \sigma_f = \tau_f, \text{ logic }]\!\} \\
& \exists \sigma_f, \sigma_I, \tau_1, \tau_2, \tau_f', \tau_I.\ \ \sigma_1 = \sigma_f \bullet \sigma_I \wedge \sigma_f \in f \wedge \\
& \sigma_1 \bullet \sigma_2 = \tau_1 \bullet \tau_2 \wedge \ \ \tau_1 = \sigma_f \bullet \tau_I \wedge \sigma' = \tau_f' \bullet \tau_I \bullet \tau_2 \\
\Rightarrow & \quad \{\![\text{ cancellativity implies } \sigma_I \bullet \sigma_2 = \tau_I \bullet \tau_2, \text{ logic }]\!\} \\
& \exists \sigma_f, \sigma_I, \tau_f'.\ \ \sigma_1 = \sigma_f \bullet \sigma_I \wedge \sigma_f \in f \ \wedge \sigma' = (\tau_f' \bullet \sigma_I) \bullet \sigma_2 \\
\Leftrightarrow & \quad \{\![\text{ definition of } ; , * \text{ and Theorem 4.5.3 }]\!\} \\
& \exists \sigma_1'.\ \ \sigma_1 \ \Delta f \ \sigma_1' \wedge \sigma' = \sigma_1' \bullet \sigma_2 \,.
\end{array}
$$

□

Proof of Lemma 5.2.5.

(a) First, $\lceil a \leq \mathit{reach}(\lceil a, a)$ by the reach induction rule from Section 5.1. Second, by a domain property, $\overline{a} = \overline{(\lceil a \cdot a)} = \langle a | \lceil a \leq \mathit{reach}(\lceil a, a)$.

(b) For $(\leq)$ we know by diamond star induction that $\mathit{reach}(\lceil a, a + b) \leq \overline{\overline{a}} \Leftarrow \lceil a \leq \overline{\overline{a}} \wedge \langle (a + b) | \overline{\overline{a}} \leq \overline{\overline{a}}$. $\lceil a \leq \overline{\overline{a}}$ holds by definition of $\overline{\ }$, while $\langle (a + b) | \overline{\overline{a}} \leq \overline{\overline{a}}$ resolves by diamond distributivity to $\langle a | \overline{\overline{a}} \leq \overline{\overline{a}} \ \wedge \ \langle b | \overline{\overline{a}} \leq \overline{\overline{a}}$. Finally, the claim holds by $\overline{(\overline{\overline{a}} \cdot a)} \leq \overline{a}$ and the assumption. The direction $(\geq)$ follows from Part a, $a \leq a + b$ and isotony of *reach*.

□

Proof of Lemma 5.3.4. We first show the auxiliary result

$$p \leq \overline{a} \ \wedge \ |a\rangle p = 0 \ \Rightarrow \ p = 0 \,. \tag{A.4}$$

We have, by the definition of diamond, full strictness of domain and since $\neg\overline{a}^{\urcorner}$ is the greatest right annihilator, $|a\rangle p = 0 \;\Leftrightarrow\; \ulcorner(a \cdot p) = 0 \;\Leftrightarrow\; a \cdot p = 0 \;\Leftrightarrow\; p \leq \neg\overline{a}^{\urcorner}$. Since by assumption $p \leq \overline{a}^{\urcorner}$, we get $p \leq \overline{a}^{\urcorner} \cdot \neg\overline{a}^{\urcorner} = 0$.

Now we continue with the proof of Lemma 5.3.4. Suppose $\overline{a}^{\urcorner} = 0$. Then by full strictness also $a = 0$ and hence $\ulcorner a = 0$, contradicting atomicity of $\ulcorner a$. Hence $\overline{a}^{\urcorner} \neq 0$.

Now assume $p \leq \overline{a}^{\urcorner} \wedge p \neq 0$. By Equation A.4 we have $0 \neq |a\rangle p = \ulcorner(a \cdot p) \leq \ulcorner a$. Hence, atomicity of $\ulcorner a$ implies $|a\rangle p = \ulcorner a$. Now, by definition of codomain and determinacy of a,

$$\overline{a}^{\urcorner} = \langle a|\ulcorner a = \langle a||a\rangle p \leq p\,,$$

so that altogether we have $p = \overline{a}^{\urcorner}$, which, by the assumptions and the definition of atomicity, shows the claim. □

Proof of Lemma 5.5.3. Assume $a_1 \in P(\mathsf{i}) \wedge a_2 \in Q(\mathsf{j}) \wedge a_3 \in R(\mathsf{k})$ and assume $a_i \neq \Box$.

(a): We only show the $\subseteq$-direction, since $\supseteq$ was already shown in Lemma 5.4.12. By the definitions it remains to show that $(a_1 + a_2) \triangleright a_3 \wedge a_1 \triangleright a_2$ implies $a_1 \triangleright (a_2 + a_3) \wedge a_2 \triangleright a_3$. The assumption $(a_1 + a_2) \triangleright a_3$ resolves to

$$\ulcorner a_1 \cdot \overline{a_3}^{\urcorner} \leq 0 \wedge \ulcorner a_2 \cdot \overline{a_3}^{\urcorner} \leq 0 \wedge a_1^{\urcorner} \cdot a_3^{\urcorner} \leq \Box \wedge a_2^{\urcorner} \cdot a_3^{\urcorner} \leq \Box \wedge a_1^{\urcorner} \cdot \ulcorner a_3 + a_2^{\urcorner} \cdot \ulcorner a_3 = \mathit{root}(a_3)\,. \quad (*)$$

The last conjunct implies $a_2^{\urcorner} \cdot \ulcorner a_3 \leq \mathit{root}(a_3)$. Moreover, note that the side condition of (a) implies $\mathit{root}(a_3) \leq \overline{a_2}^{\urcorner}$. Hence, $\mathit{root}(a_3) = \mathit{root}(a_3) \cdot \ulcorner a_3 \leq \overline{a_2}^{\urcorner} \cdot \ulcorner a_3 = \ulcorner a_2 \cdot \ulcorner a_3 + a_2^{\urcorner} \cdot \ulcorner a_3 = a_2^{\urcorner} \cdot \ulcorner a_3$ and therefore $\mathit{root}(a_3) = a_2^{\urcorner} \cdot \ulcorner a_3$. This shows $a_2 \triangleright a_3$, which further by Lemma 5.4.3 implies $\mathit{root}(a_2 + a_3) = \mathit{root}(a_2)$ and $a_1^{\urcorner} \cdot \ulcorner a_3 \leq \mathit{root}(a_3)$. From this we obtain by $(*)$, since $\mathit{root}(a_3) \neq \Box$ is an atom and $a_1^{\urcorner} \cdot a_2^{\urcorner} \leq \Box$ by $a_1 \triangleright a_2$, that $a_1^{\urcorner} \cdot \ulcorner a_3 = 0$ as well. Hence, again by $a_1 \triangleright a_2$, we obtain $\mathit{root}(a_2) = a_1^{\urcorner} \cdot \ulcorner a_2 + a_1^{\urcorner} \cdot \ulcorner a_3$, which establishes $a_1 \triangleright (a_2 + a_3)$.

(b): The $\subseteq$-direction was again shown in Lemma 5.4.12. Now assume $a_1 \triangleright (a_2 + a_3)$ and $a_2 \mathbin{\#} a_3$. The side condition implies $\overline{a_1}^{\urcorner} \cdot \mathit{root}(a_3) \leq 0$ which in turn implies $a_1^{\urcorner} \cdot \ulcorner a_3 \leq \neg \mathit{root}(a_3)$. Therefore $a_1 \triangleright a_3$ does not hold and consequently $a_1 \triangleright a_2$ and $a_1 \mathbin{\#} a_3$ need to be true by the definition of $\triangleright$ for forests.

(c): We assume $(a_1 + a_2) \triangleright a_3 \wedge a_1 \triangleright a_2$ and show $a_1 \triangleright (a_2 + a_3) \wedge a_2 \mathbin{\#} a_3$. As for (a), $(a_1 + a_2) \triangleright a_3$ implies $a_1^{\urcorner} \cdot \ulcorner a_3 + a_2^{\urcorner} \cdot \ulcorner a_3 = \mathit{root}(a_3)$. We calculate $a_2^{\urcorner} \cdot \ulcorner a_3 \leq a_2^{\urcorner} \cdot \mathit{root}(a_3) = a_2^{\urcorner} \cdot \overline{a_1}^{\urcorner} \cdot \mathit{root}(a_3) = a_2^{\urcorner} \cdot a_1^{\urcorner} \cdot \mathit{root}(a_3) \leq \Box \cdot \ulcorner a_3 \leq 0$ by assumptions and the side condition. Hence, $a_2 \triangleright a_3$ and $a_1^{\urcorner} \cdot \ulcorner a_3 = \mathit{root}(a_3)$ which by the assumption $(a_1 + a_2) \triangleright a_3$ further implies $a_1 \triangleright a_3$. Next, the reverse direction is shown by $\mathit{root}(a_i) \leq \overline{a_1}^{\urcorner} \Rightarrow \neg(a_1 \mathbin{\#} a_i)$, which in turn implies by $a_1 \triangleright (a_2 + a_3)$ and Definition 5.4.4 that $a_1 \triangleright a_i$ for $i = 2, 3$. Now, using assumption $a_2 \triangleright a_3$ we immediately get $(a_1 + a_2) \triangleright a_3$ from Definition 5.4.4 again.

(d): Again $\supseteq$ was proved in Lemma 5.4.12 while $\subseteq$ holds, since the side condition implies $root(a_3) \leq \overline{a_2}$ and hence $a_1 \triangleright a_3$ can not hold by $a_1 \mathbin{⋕} a_2$. Therefore by definition we can only have $a_1 \mathbin{⋕} a_3 \wedge a_2 \triangleright a_3$. Now the claim follows from bilinearity of $⋕$. □

Proof of Equation (5.3) $\Leftrightarrow$ Equation (5.4).

$$
\begin{aligned}
& \mathsf{cell}\,(\mathsf{i}) \circledcirc (\mathsf{tree}(\mathsf{i.left}) \circledast (\mathsf{cell}\,(\mathsf{i.right}) \circledcirc (\mathsf{tree}(\mathsf{i.right.left}) \circledast \mathsf{tree}(\mathsf{i.right.right})))) \\
= \quad & \{\![\text{ Lemma 5.5.3 (c) }]\!\} \\
& (\mathsf{cell}\,(\mathsf{i}) \circledcirc \mathsf{tree}(\mathsf{i.left})) \circledcirc (\mathsf{cell}\,(\mathsf{i.right}) \circledcirc (\mathsf{tree}(\mathsf{i.right.left}) \circledast \mathsf{tree}(\mathsf{i.right.right}))) \\
= \quad & \{\![\text{ definition of } \mathsf{rt\_context}]\!\} \\
& \mathsf{rt\_context}(\mathsf{i}) \circledcirc (\mathsf{cell}\,(\mathsf{i.right}) \circledcirc (\mathsf{tree}(\mathsf{i.right.left}) \circledast \mathsf{tree}(\mathsf{i.right.right}))) \\
= \quad & \{\![\text{ commutativity of } \circledast]\!\} \\
& \mathsf{rt\_context}(\mathsf{i}) \circledcirc (\mathsf{cell}\,(\mathsf{i.right}) \circledcirc (\mathsf{tree}(\mathsf{i.right.right}) \circledast \mathsf{tree}(\mathsf{i.right.left}))) \\
= \quad & \{\![\text{ Lemma 5.5.3 (c) }]\!\} \\
& \mathsf{rt\_context}(\mathsf{i}) \circledcirc ((\mathsf{cell}\,(\mathsf{i.right}) \circledcirc \mathsf{tree}(\mathsf{i.right.right})) \circledcirc \mathsf{tree}(\mathsf{i.right.left})) \\
= \quad & \{\![\text{ definition of } \mathsf{lt\_context}]\!\} \\
& \mathsf{rt\_context}(\mathsf{i}) \circledcirc (\mathsf{lt\_context}(\mathsf{i.right}) \circledcirc \mathsf{tree}(\mathsf{i.right.left}))
\end{aligned}
$$

The same calculation can be done for the final state of Figure 5.7, i.e., the equation

$$\mathsf{cell}\,(\mathsf{j}) \circledcirc ((\mathsf{cell}\,(\mathsf{i,j.left}) \circledcirc (\mathsf{tree}(\mathsf{i.left}) \circledast \mathsf{tree}(\mathsf{k,i.right}))) \circledast \mathsf{tree}(\mathsf{j.right}))$$

equals the following

$$\mathsf{lt\_context}(\mathsf{j}) \circledcirc (\mathsf{rt\_context}(\mathsf{i,j.left}) \circledcirc \mathsf{tree}(\mathsf{k,i.right}))\,.$$

□

A.2 Further Properties of the Assertion Calculus

We provide some theorems that characterise in particular the interplay of pure and precise assertions in combination with residuals and detachment operators. By this we also demonstrate simple algebraic proofs of non-trivial properties characterising behaviour of assertions in separation logic.

Lemma A.2.1 *If t is a test then the element $t \cdot \top$ is pure.*

Proof. We use the characterisation of Definition 3.2.6 and calculate $(t \cdot \top \sqcap 1) \cdot \top = t \cdot (\top \sqcap 1) \cdot \top = t \cdot \top$ which follows immediately from the equation (testdist). □

We now investigate the interplay between pure assertions and algebraic residuals.

Corollary A.2.2 *For arbitrary elements a, b, c we have $a \sqcap (b\backslash c) \leq (a \sqcap b)\backslash c$.*

Proof. By Definition 3.1.3, isotony and Lemma 3.1.5,

$$\begin{aligned} a \sqcap (b\backslash c) \leq (a \sqcap b)\backslash c &\Leftrightarrow (a \sqcap b) \cdot (a \sqcap (b\backslash c)) \leq c \\ &\Leftarrow b \cdot (b\backslash c) \leq c \\ &\Leftarrow c \leq c \\ &\Leftrightarrow \mathsf{true}\,. \end{aligned}$$

□

Lemma A.2.3 *If a is pure then for arbitrary b, c the following (in)equations hold:*

$$(a \sqcap b)\backslash(a \sqcap c) = (a \sqcap b)\backslash c\,, \tag{A.5}$$

$$a \sqcap (b\backslash c) \leq b\backslash(a \sqcap c)\,, \tag{A.6}$$

$$a \sqcap (b\backslash c) = a \sqcap (b\backslash(a \sqcap c))\,, \tag{A.7}$$

$$a \sqcap (b\backslash c) = a \sqcap ((a \sqcap b)\backslash c)\,, \tag{A.8}$$

$$a \sqcap (b\backslash c) = a \sqcap ((a \sqcap b)\backslash(a \sqcap c))\,, \tag{A.9}$$

$$a \sqcap ((a \sqcap b)\backslash(a \sqcap c)) \leq b\backslash(a \sqcap c)\,. \tag{A.10}$$

Proof. For a proof of (A.5) we use the proof principle of indirect equality, i.e.,

$$x = y \;\Leftrightarrow\; (\forall\, z :\; z \leq x \Leftrightarrow z \leq y)\,.$$

The $\Rightarrow$-direction is obvious while for the converse we can instantiate z to x and y and hence by antisymmetry of $\leq$ the claim holds. Next we continue with the main proof. By definition, Lemma 3.2.7(c), shunting, distributivity, $a + \overline{a} = \top$, shunting, Lemma 3.2.7(c), and definition:

$$\begin{aligned} & \forall\, x :\; x \leq (a \sqcap b)\backslash(a \sqcap c) \\ \Leftrightarrow\; & \forall\, x :\; (a \sqcap b) \cdot x \leq a \sqcap c \\ \Leftrightarrow\; & \forall\, x :\; a \sqcap b \cdot x \leq a \sqcap c \\ \Leftrightarrow\; & \forall\, x :\; b \cdot x \leq \overline{a} + (a \sqcap c) \\ \Leftrightarrow\; & \forall\, x :\; b \cdot x \leq (a + \overline{a}) \sqcap (\overline{a} + c) \\ \Leftrightarrow\; & \forall\, x :\; a \sqcap b \cdot x \leq c \\ \Leftrightarrow\; & \forall\, x :\; (a \sqcap b) \cdot x \leq c \\ \Leftrightarrow\; & \forall\, x :\; x \leq (a \sqcap b)\backslash c\,. \end{aligned}$$

Now we give a proof of Part (A.6). By definition of residuals, Lemma 3.2.7(c), Lemma 3.1.5 and isotony,

$$a \sqcap (b\backslash c) \leq b\backslash(a \sqcap c) \;\Leftrightarrow\; b \cdot (a \sqcap (b\backslash c)) \leq a \sqcap c \;\Leftrightarrow\; a \sqcap (b \cdot (b\backslash c)) \leq a \sqcap c \;\Leftrightarrow\; \mathsf{true}\,.$$

The $\leq$-direction of (A.7) follows immediately from (A.6) by multiplying both side with $a\sqcap$ and using idempotence of $\sqcap$. The $\geq$-direction can be shown as follows: By idempotence and isotony of $\sqcap$, definition of resdiduals, Lemma 3.2.7(c), Lemma 3.1.5, and isotony,

$$\begin{aligned}
& a \sqcap (b\backslash(a \sqcap c)) \leq a \sqcap (b\backslash c) \\
\Leftrightarrow\; & a \sqcap (b\backslash(a \sqcap c)) \leq b\backslash c \\
\Leftrightarrow\; & b \cdot (a \sqcap (b\backslash(a \sqcap c))) \leq c \\
\Leftrightarrow\; & a \sqcap (b \cdot (b\backslash(a \sqcap c))) \leq c \\
\Leftrightarrow\; & \mathsf{true}\,.
\end{aligned}$$

Next, the $\leq$-direction of (A.8) follows immediately by Corollary A.2.2. By idempotence and isotony of $\sqcap$, definition of residuals, Lemma 3.2.7(c), and Lemma 3.1.5:

$$\begin{aligned}
& a \sqcap ((a \sqcap b)\backslash c) \leq a \sqcap (b\backslash c) \\
\Leftrightarrow\; & a \sqcap ((a \sqcap b)\backslash c) \leq b\backslash c \\
\Leftrightarrow\; & b \cdot (a \sqcap ((a \sqcap b)\backslash c)) \leq c \\
\Leftrightarrow\; & (a \sqcap b) \cdot ((a \sqcap b)\backslash c) \leq c \\
\Leftrightarrow\; & \mathsf{true}\,.
\end{aligned}$$

Finally (A.9) follows immediately from (A.8) and (A.5) and Equation (A.10) holds by (A.9) and (A.6). □

These laws provided frequently used theorems characterising the interplay between pure elements and residuals. It can be seen that all of them can be calculated in a simple and purely algebraic fashion. We now turn to detachments interacting with pure elements. This yields analogous import and export laws for detachments as in the case of pure elements interacting with multiplication or more concretely separating conjunction in Lemma 3.2.7(c).

Lemma A.2.4 *Assume a is pure and b, c arbitrarily. Then* $(a \sqcap c)\rfloor(a \sqcap b) = c\rfloor(a \sqcap b)$.

Proof. The $\leq$-direction follow immediately from isotony of $\rfloor$ in its first argument. The other direction can be shown as follows: By (exc), definition of $\rfloor$, Boolean algebra, shunting, Lemma 3.2.7(c), Lemma A.2.3 (A.7) and (A.9), Lemma 3.2.7(c), shunting, Boolean algebra, commutativity of $\cdot$ and Lemma 3.1.5,

$$
\begin{array}{ll}
& c\rfloor(a \sqcap b) \leq (a \sqcap c)\rfloor(a \sqcap b) \\
\Leftrightarrow & c \cdot \overline{(a \sqcap c)\rfloor(a \sqcap b)} \leq \overline{a \sqcap b} \\
\Leftrightarrow & c \cdot ((a \sqcap c)\backslash\overline{a \sqcap b}) \leq \overline{a \sqcap b} \\
\Leftrightarrow & c \cdot ((a \sqcap c)\backslash\overline{a \sqcap b}) \leq \overline{a} + \overline{b} \\
\Leftrightarrow & a \sqcap (c \cdot ((a \sqcap c)\backslash\overline{a \sqcap b})) \leq \overline{b} \\
\Leftrightarrow & c \cdot (a \sqcap ((a \sqcap c)\backslash\overline{a \sqcap b})) \leq \overline{b} \\
\Leftrightarrow & c \cdot (a \sqcap (c\backslash\overline{a \sqcap b})) \leq \overline{b} \\
\Leftrightarrow & a \sqcap (c \cdot (c\backslash\overline{a \sqcap b})) \leq \overline{b} \\
\Leftrightarrow & c \cdot (c\backslash\overline{a \sqcap b}) \leq \overline{a} + \overline{b} \\
\Leftrightarrow & c \cdot (c\backslash\overline{a \sqcap b}) \leq \overline{a \sqcap b} \\
\Leftrightarrow & \mathsf{true}\,.
\end{array}
$$

□

Lemma A.2.5 *For arbitrary b, c and pure a the inequation $(a \sqcap c)\rfloor(\overline{a} \sqcap b) \leq 0$ is valid. Therefore also $(\overline{a} \sqcap c)\rfloor(a \sqcap b) \leq 0$.*

Proof. By (exc), Boolean algebra, Lemma 3.2.7(c), and isotony:

$$
\begin{array}{ll}
& (\overline{a} \sqcap b)\lfloor(a \sqcap c) \leq 0 \\
\Leftrightarrow & \top \cdot (a \sqcap c) \leq \overline{\overline{a} \sqcap b} \\
\Leftrightarrow & \top \cdot (a \sqcap c) \leq a + \overline{b} \\
\Leftrightarrow & a \sqcap \top \cdot c \leq a + \overline{b} \\
\Leftrightarrow & \mathsf{true}\,.
\end{array}
$$

□

Lemma A.2.6 *For arbitrary b, c and pure a we have $(a \sqcap c)\rfloor b = c\rfloor(a \sqcap b)$.*

Proof. By Boolean algebra, distributivity of $\rfloor$, Lemma A.2.5, and Lemma A.2.4:

$$
\begin{array}{ll}
& (a \sqcap c)\rfloor b \\
= & (a \sqcap c)\rfloor((a \sqcap b) + (\overline{a} \sqcap b)) \\
= & ((a \sqcap c)\rfloor(a \sqcap b)) + ((a \sqcap c)\rfloor(\overline{a} \sqcap b)) \\
= & (a \sqcap c)\rfloor(a \sqcap b) \\
= & c\rfloor(a \sqcap b)\,.
\end{array}
$$

□

Lemma A.2.7 *For arbitrary b, c and pure a we have $a \sqcap (c\rfloor b) = c\rfloor(a \sqcap b)$.*

Proof. We first show the $\geq$-direction: By (exc), Boolean algebra, distributivity, supremum splitting and isotony, isotony of $\cdot$ and definition of $\rfloor$, commutativity, Lemma 3.2.7(a) and Lemma 3.1.5,

$$
\begin{array}{ll}
& c\rfloor(a \sqcap b) \leq a \sqcap (c\rfloor b) \\
\Leftrightarrow & c \cdot \overline{a \sqcap (c\rfloor b)} \leq \overline{a \sqcap b} \\
\Leftrightarrow & c \cdot (\overline{a} + \overline{c\rfloor b}) \leq \overline{a} + \overline{b} \\
\Leftrightarrow & (c \cdot \overline{a}) + (c \cdot \overline{c\rfloor b}) \leq \overline{a} + \overline{b} \\
\Leftarrow & c \cdot \overline{a} \leq \overline{a} \;\; \wedge \;\; c \cdot \overline{c\rfloor b} \leq \overline{b} \\
\Leftarrow & \top \cdot \overline{a} \leq \overline{a} \;\; \wedge \;\; c \cdot (c\backslash \overline{b}) \leq \overline{b} \\
\Leftrightarrow & \mathsf{true}\,.
\end{array}
$$

For the other direction we calculate: By shunting, (exc), Boolean algebra, superdistributivity, Lemma 3.1.5, Boolean algebra,

$$
\begin{array}{ll}
& a \sqcap (c\rfloor b) \leq c\rfloor(a \sqcap b) \\
\Leftrightarrow & c\rfloor b \leq c\rfloor(a \sqcap b) + \overline{a} \\
\Leftrightarrow & c \cdot \overline{c\rfloor(a \sqcap b) + \overline{a}} \leq \overline{b} \\
\Leftrightarrow & c \cdot ((c\backslash \overline{a \sqcap b}) \sqcap a) \leq \overline{b} \\
\Leftarrow & c \cdot (c\backslash \overline{a \sqcap b}) \sqcap c \cdot a \leq \overline{b} \\
\Leftarrow & \overline{a \sqcap b} \sqcap c \cdot a \leq \overline{b} \\
\Leftrightarrow & (\overline{a} \sqcap c \cdot a) + (\overline{b} \sqcap c \cdot a) \leq \overline{b}\,.
\end{array}
$$

Now, since a is pure, we infer $\overline{a} \sqcap c \cdot a \leq \overline{a} \sqcap \top \cdot a \leq \overline{a} \sqcap a \leq 0$ and claim follows from isotony. □

Corollary A.2.8 *For arbitrary b, c and pure a,*

$$a \sqcap c\rfloor b = c\rfloor(a \sqcap b) = (a \sqcap c)\rfloor b = (a \sqcap c)\rfloor(a \sqcap b)\,.$$

As a next step we give some further useful properties of precise assertions which facilitates calculating with them.

Lemma A.2.9 *If b, c are precise and a is pure, then $(a \sqcap b) + (\overline{a} \sqcap c)$ is precise.*

Proof. We assume arbitrary elements $d, e \in S$. By distributivity, Lemma 3.2.7(c) and Corollary 3.2.11, distributivity, $a \sqcap \overline{a} = 0$, idempotence of $\sqcap$, b and c are precise, Lemma 3.2.7(c) and Corollary 3.2.11, and distributivity:

$$
\begin{array}{ll}
& ((a \sqcap b) + (\overline{a} \sqcap c)) \cdot d \sqcap ((a \sqcap b) + (\overline{a} \sqcap c)) \cdot e \\
= & ((a \sqcap b) \cdot d + (\overline{a} \sqcap c) \cdot d) \sqcap ((a \sqcap b) \cdot e + (\overline{a} \sqcap c) \cdot e) \\
= & (a \sqcap (b \cdot d) + \overline{a} \sqcap (c \cdot d)) \sqcap (a \sqcap (b \cdot e) + \overline{a} \sqcap (c \cdot e)) \\
= & a \sqcap (b \cdot d) \sqcap a \sqcap (b \cdot e) + \overline{a} \sqcap (c \cdot d) \sqcap a \sqcap (b \cdot e) + \\
& a \sqcap (b \cdot d) \sqcap \overline{a} \sqcap (c \cdot e) + \overline{a} \sqcap (c \cdot d) \sqcap \overline{a} \sqcap (c \cdot e)
\end{array}
$$

$$
\begin{array}{ll}
= & a \sqcap (b \cdot d) \sqcap (b \cdot e) + \overline{a} \sqcap (c \cdot d) \sqcap (c \cdot e) \\
= & a \sqcap (b \cdot (d \sqcap e)) + \overline{a} \sqcap (c \cdot (d \sqcap e)) \\
= & ((a \sqcap b) \cdot (d \sqcap e)) + ((\overline{a} \sqcap c) \cdot (d \sqcap e)) \\
= & ((a \sqcap b) + (\overline{a} \sqcap c)) \cdot (d \sqcap e)\,.
\end{array}
$$

□

Lemma A.2.10 *If a is precise then $(a \sqcap b) \cdot c \sqcap a \cdot d = (a \sqcap b) \cdot (c \sqcap d)$ for arbitrary elements b, c, d.*

Proof. First, we calculate by isotony, Boolean algebra, and a is precise,

$$(a \sqcap b) \cdot (c \sqcap \overline{d}) \sqcap a \cdot d \;\leq\; a \cdot (c \sqcap \overline{d}) \sqcap a \cdot d = a \cdot ((c \sqcap \overline{d}) \sqcap d) = 0\,. \qquad (*)$$

We prove the equation by showing each inequation separately. The $\leq$-direction can be shown as follows: By Boolean algebra, distributivity, by $(*)$, and isotony:

$$
\begin{array}{ll}
& (a \sqcap b) \cdot c \sqcap a \cdot d \\
= & (((a \sqcap b) \cdot (c \sqcap d)) + ((a \sqcap b) \cdot (c \sqcap \overline{d}))) \sqcap a \cdot d \\
= & (((a \sqcap b) \cdot (c \sqcap d) \sqcap a \cdot d) + ((a \sqcap b) \cdot (c \sqcap \overline{d}) \sqcap a \cdot d) \\
= & ((a \sqcap b) \cdot (c \sqcap d)) \sqcap (a \cdot d) \\
\leq & (a \sqcap b) \cdot (c \sqcap d)\,.
\end{array}
$$

The converse inequation follows by isotony, i.e., $(a \sqcap b) \cdot (c \sqcap d) \leq (a \sqcap b) \cdot c \sqcap (a \sqcap b) \cdot d \leq (a \sqcap b) \cdot c \sqcap a \cdot d$. □

Corollary A.2.11 *If a or a' is precise, then $(a \sqcap b) \cdot a' \sqcap a \cdot (a' \sqcap c) = (a \sqcap b) \cdot (a' \sqcap c)$ for all b, c.*

This law characterises an interplay between $\cdot$ and $\sqcap$ w.r.t. precise elements in the sense that only the subheaps that fit together, i.e., a and $a \sqcap b$, respectively a' and $a' \sqcap c$, remain within the intersection on the left-hand side of the above equation.

A.3 Deferred Figures

$$
\begin{array}{ll}
\mathrm{FV}(\, v := e \,) & =_{df} \{v\} \cup \mathrm{FV}(e) \\
\mathrm{FV}(\, \mathsf{skip} \,) & =_{df} \emptyset \\
\mathrm{FV}(P \,;\, Q \,) & =_{df} \mathrm{FV}(P) \cup \mathrm{FV}(Q) \\
\mathrm{FV}(\, \mathsf{if}\ b\ \mathsf{then}\ P\ \mathsf{else}\ Q \,) & =_{df} \mathrm{FV}(b) \cup \mathrm{FV}(P) \cup \mathrm{FV}(Q) \\
\mathrm{FV}(\, \mathsf{while}\ b\ \mathsf{do}\ P \,) & =_{df} \mathrm{FV}(b) \cup \mathrm{FV}(P) \\
\mathrm{FV}(\, \mathsf{newvar}\ v\ \mathsf{in}\ P \,) & =_{df} \mathrm{FV}(P) - \{v\} \\
\mathrm{FV}(\, \mathsf{newvar}\ v := e\ \mathsf{in}\ P \,) & =_{df} (\mathrm{FV}(e) \cup \mathrm{FV}(P)) - \{v\} \\
\mathrm{FV}(\, v := \mathsf{cons}\,(e_1, ..., e_n) \,) & =_{df} \{v\} \cup \bigcup_{i=1}^{n} \mathrm{FV}(e_i) \\
\mathrm{FV}(\, v := [e] \,) & =_{df} \{v\} \cup \mathrm{FV}(e) \\
\mathrm{FV}(\, [e_1] := e_2 \,) & =_{df} \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2) \\
\mathrm{FV}(\, \mathsf{dispose}\ e \,) & =_{df} \mathrm{FV}(e)
\end{array}
$$

Figure A.1: Definition of the free variables FV(_) of syntactical commands.

$$
\begin{array}{ll}
\mathrm{MV}(\, v := e \,) & =_{df} \{v\} \\
\mathrm{MV}(\, \mathsf{skip} \,) & =_{df} \emptyset \\
\mathrm{MV}(P \,;\, Q \,) & =_{df} \mathrm{MV}(P) \cup \mathrm{MV}(Q) \\
\mathrm{MV}(\, \mathsf{if}\ b\ \mathsf{then}\ P\ \mathsf{else}\ Q \,) & =_{df} \mathrm{MV}(P) \cup \mathrm{MV}(Q) \\
\mathrm{MV}(\, \mathsf{while}\ b\ \mathsf{do}\ P \,) & =_{df} \mathrm{MV}(P) \\
\mathrm{MV}(\, \mathsf{newvar}\ v\ \mathsf{in}\ P \,) & =_{df} \mathrm{MV}(P) - \{v\} \\
\mathrm{MV}(\, \mathsf{newvar}\ v := e\ \mathsf{in}\ P \,) & =_{df} \mathrm{MV}(P) - \{v\} \\
\mathrm{MV}(\, v := \mathsf{cons}\,(e_1, ..., e_n) \,) & =_{df} \{v\} \\
\mathrm{MV}(\, v := [e] \,) & =_{df} \{v\} \\
\mathrm{MV}(\, [e_1] := e_2 \,) & =_{df} \emptyset \\
\mathrm{MV}(\, \mathsf{dispose}\ e \,) & =_{df} \emptyset
\end{array}
$$

Figure A.2: Definition of the modified variables MV(_) of syntactical commands.

Bibliography

[AS12] A. Armstrong and G. Struth, *Automated Reasoning in Higher-Order Regular Algebra*, Relational and Algebraic Methods in Computer Science (W. Kahl and T. G. Griffin, eds.), Lecture Notes in Computer Science, vol. 7560, Springer-Verlag, 2012, pp. 66–81.

[ASW13a] A. Armstrong, G. Struth, and T. Weber, *Kleene Algebra*, Archive of Formal Proofs, 2013.

[ASW13b] ———, *Program Analysis and Verification Based on Kleene Algebra in Isabelle/HOL*, Interactive Theorem Proving (S. Blazy, C. Paulin-Mohring, and D. Pichardie, eds.), Lecture Notes in Computer Science, vol. 7998, Springer-Verlag, 2013, pp. 197–212.

[BBTS05] B. Biering, L. Birkedal, and N. Torp-Smith, *BI Hyperdoctrines and Higher-Order Separation Logic*, Proccedings of 14th European Symposium on Programming (ESOP 2005) (S. Sagiv, ed.), Lecture Notes in Computer Science, vol. 3444, Springer-Verlag, 2005, pp. 233–247.

[BBTS07] ———, *BI-hyperdoctrines, Higher-order Separation Logic, and Abstraction*, ACM Transactions on Programming Languages and Systems **29** (2007), no. 5, 24.

[BCO05] J. Berdine, C. Calcagno, and P. O'Hearn, *A Decidable Fragment of Separation Logic*, FSTTCS 2004: Foundations of Software Technology and Theoretical Computer Science **3328** (2005), 110–117.

[BCO06] J. Berdine, C. Calcagno, and P. W. O'Hearn, *Smallfoot: Modular Automatic Assertion Checking with Separation Logic*, Formal Methods for Components and Objects (FMCO2005) (F. de Boer, M. M. Bonsangue, S. Graf, and W. de Roever, eds.), Lecture Notes in Computer Science, vol. 4111, Springer-Verlag, 2006, pp. 115–137.

[BCOP05] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson, *Permission Accounting in Separation Logic*, Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, Principles of Programming Languages POPL'05, ACM Press, 2005, pp. 259–270.

[BCY06] R. Bornat, C. Calcagno, and H. Yang, *Variables As Resource in Separation Logic*, Electronic Notes on Theoretical Computer Science **155** (2006), 247–276.

[BD02] M. Broy and E. Denert, *Software Pioneers: Contributions to Software Engineering*, Springer-Verlag, 2002.

[Bie04] B. Biering, *On the Logic of Bunched Implications and its Relation to Separation Logic*, Master's thesis, University of Copenhagen, 2004.

[Bir67] G. Birkhoff, *Lattice Theory*, 3rd ed., American Mathematical Society, 1967.

[BJ72] T. S. Blyth and M. F. Janowitz, *Residuation Theory*, Pergamon Press, 1972.

[BK10] J. Brotherston and M. Kanovich, *Undecidability of Propositional Separation Logic and Its Neighbours*, Proceedings of the 2010 25th Annual IEEE Symposium on Logic in Computer Science, LICS '10, IEEE Computer Society, 2010, pp. 130–139.

[Boy03] J. Boyland, *Checking Interference with Fractional Permissions*, Proceedings of the 10th International Conference on Static Analysis, SAS'03, Springer-Verlag, 2003, pp. 55–72.

[BP12] T. Braibant and D. Pous, *Deciding Kleene Algebras in Coq*, Logical Methods in Computer Science **8** (2012), no. 1, 1–42.

[Bro07] S. Brookes, *A Semantics for Concurrent Separation Logic*, Theoretical Computer Science **375** (2007), 227–270.

[Bur72] R. M. Burstall, *Some Techniques for Proving Correctness of Programs which Alter Data Structures*, Machine Intelligence **7** (1972), 23–50.

[BV14] J. Brotherston and J. Villard, *Parametric Completeness for Separation Theories*, Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, ACM Press, 2014, pp. 453–464.

[BvdW93] R. C. Backhouse and J. van der Woude, *Demonic Operators and Monotype Factors*, Mathematical Structures in Computer Science **3** (1993), no. 4, 417–433.

[BvW98] R.-J. Back and J. von Wright, *Refinement Calculus: A Systematic Introduction*, Graduate Texts in Computer Science, Springer-Verlag, 1998.

[CDOY09a] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang, *Compositional Shape Analysis by Means of Bi-abduction*, ACM SIGPLAN Notices **44** (2009), no. 1, 289–300.

[CDOY09b] ———, *Space Invading Systems Code*, Logic-Based Program Synthesis and Transformation (M. Hanus, ed.), Springer-Verlag, 2009, pp. 1–3.

[CJ00] P. Collette and C. B. Jones, *Enhancing the Tractability of Rely/Guarantee Specifications in the Development of Interfering Operations*, Proof, Language and Interaction (G. Plotkin, C. Stirling, and M. Tofte, eds.), MIT Press, 2000, pp. 277–307.

[Coh94] E. Cohen, *Using Kleene Algebra to Reason about Concurrency Control*, Tech. report, Telcordia, 1994.

[Con71] J. H. Conway, *Regular Algebra and Finite Machines*, Chapman & Hall, 1971.

[COY07] C. Calcagno, P. W. O'Hearn, and H. Yang, *Local Action and Abstract Separation Logic*, Proceedings of the 22nd Symposium on Logic in Computer Science, IEEE Computer Society Press, 2007, pp. 366–378.

[CS10] Y. Chen and J.-W. Sanders, *Abstraction of Object Graphs in Program Verification*, Proc. of 10th Intl. Conference on Mathematics of Program Construction (C. Bolduc, J. Desharnais, and B. Ktari, eds.), Lecture Notes in Computer Science, vol. 6120, Springer-Verlag, 2010, pp. 80–99.

[Dan09] H.-H. Dang, *Algebraic Aspects of Separation Logic*, Tech. Report 2009-1, Institut für Informatik, Universität Augsburg, 2009.

[Dan12] ———, *On the Algebraic Derivation of Garbage Collectors*, Relational and Algebraic Methods in Computer Science — PhD Programme at RAMiCS 13 (W. Kahl and T. G. Griffin, eds.), Technical Report, Faculty of Computer Science and Technology, University of Cambridge, 2012, http://www.cl.cam.ac.uk/conference/ramics13/Dang_alggarcol.pdf.

[Dan14] _____, *Abstract Dynamic Frames*, Relational and Algebraic Methods in Computer Science (RAMiCS 14) (P. Höfner, P. Jipsen, W. Kahl, and M. E. Müller, eds.), Lecture Notes in Computer Science, vol. 8428, Springer-Verlag, 2014, pp. 157–172.

[DBS+95] J. Desharnais, N. Belkhiter, S. B. M. Sghaier, F. Tchier, A. Jaoua, A. Mili, and N. Zaguia, *Embedding a Demonic Semilattice in a Relational Algebra*, Theoretical Computer Science **149** (1995), no. 2, 333–360.

[DGM+14] H.-H. Dang, R. Glück, B. Möller, P. Roocks, and A. Zelend, *Exploring Modal Worlds*, Journal of Logic and Algebraic Programming **83** (2014), 135–153.

[DH08] H.-H. Dang and P. Höfner, *First-Order Theorem Prover Evaluation w.r.t. Relation- and Kleene Algebra*, Relations and Kleene Algebra in Computer Science — PhD Programme at RelMiCS 10/AKA 05 (R. Berghammer, B. Möller, and G. Struth, eds.), Technical Report, no. 2008-04, Institut für Informatik, Universität Augsburg, 2008, pp. 48–52.

[DH11] _____, *Variable Side Conditions and Greatest Relations in Algebraic Separation Logic*, Proceedings of the 12th international conference on Relational and Algebraic Methods in Computer Science (H. de Swart, ed.), Lecture Notes in Computer Science, vol. 6663, Springer-Verlag, 2011, pp. 125–140.

[DH12] _____, *Automated Higher-order Reasoning about Quantales*, PAAR-2010: Proceedings of the 2nd Workshop on Practical Aspects of Automated Reasoning (R. A. Schmidt, S. Schulz, and B. Konev, eds.), EPiC Series, vol. 9, EasyChair, 2012, http://www.easychair.org/publications/?page=515428679, pp. 40–51.

[DHA09] R. Dockins, A. Hobor, and A. W. Appel, *A Fresh Look at Separation Algebras and Share Accounting*, Proceedings of the 7th Asian Symposium on Programming Languages and Systems, APLAS '09, Springer-Verlag, 2009, pp. 161–177.

[DHM09] H.-H. Dang, P. Höfner, and B. Möller, *Towards Algebraic Separation Logic*, Relations and Kleene Algebra in Computer Science (R. Berghammer, A. Jaoua, and B. Möller, eds.), Lecture Notes in Computer Science, vol. 5827, Springer-Verlag, 2009, pp. 59–72.

[DHM10] _____, *Algebraic Separation Logic*, Tech. Report 2010-06, Institute of Computer Science, University of Augsburg, 2010.

[DHM11] ———, *Algebraic Separation Logic*, Journal of Logic and Algebraic Programming **80** (2011), no. 6, 221–247.

[Dij76] E. W. Dijkstra, *A Discipline of Programming*, Prentice Hall, 1976.

[DM01a] J. Desharnais and B. Möller, *Characterizing Determinacy in Kleene Algebra*, Information Sciences **139** (2001), 253–273.

[DM01b] ———, *Characterizing Determinacy in Kleene Algebra (revised version)*, Tech. Report 2001-03, Institute of Computer Science, University of Augsburg, April 2001.

[DM11] H.-H. Dang and B. Möller, *Simplifying Pointer Kleene Algebra*, Proceedings of the First Workshop on Automated Theory Engineering, Wrocaw, Poland (P. Höfner, A. McIver, and G. Struth, eds.), CEUR Workshop Proceedings, vol. 760, CEUR-WS.org, 2011, pp. 20–29.

[DM12a] ———, *Reverse Exchange for Concurrency and Local Reasoning*, Mathematics of Program Construction (J. Gibbons and P. Nogueira, eds.), Lecture Notes in Computer Science, vol. 7342, Springer-Verlag, 2012, pp. 177–197.

[DM12b] ———, *Transitive Separation Logic*, 13th International Conference on Relational and Algebraic Methods in Computer Science (RAMiCS 13) (W. Kahl and T. G. Griffin, eds.), Lecture Notes in Computer Science, vol. 7560, Springer-Verlag, 2012, pp. 1–16.

[DM13] ———, *Extended Transitive Separation Logic*, Tech. Report 2013-07, Institut für Informatik, Universität Augsburg, 2013.

[DM14] ———, *Concurrency and Local Reasoning under Reverse Exchange*, Science of Computer Programming **85, Part B** (2014), 204–223, Special Issue on Mathematics of Program Construction 2012.

[DMN97] J. Desharnais, A. Mili, and T. T. Nguyen, *Refinement and Demonic Semantics*, Relational Methods in Computer Science (C. Brink, W. Kahl, and G. Schmidt, eds.), Springer-Verlag, 1997, pp. 166–183.

[DMS06] J. Desharnais, B. Möller, and G. Struth, *Kleene Algebra with Domain*, ACM Transactions on Computational Logic **7** (2006), no. 4, 798–833.

[DMT06] J. Desharnais, B. Möller, and F. Tchier, *Kleene Under a Modal Demonic Star*, Journal of Logic and Algebraic Programming **66** (2006), 127–160.

[DYBG+13] T. Dinsdale-Young, L. Birkedal, P. Gardner, M. Parkinson, and H. Yang, *Views: Compositional Reasoning for Concurrent Programs*, Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '13, ACM Press, 2013, pp. 287–300.

[DYDG+10] T. Dinsdale-Young, M. Dodds, P. Gardner, M. Parkinson, and V. Vafeiadis, *Concurrent Abstract Predicates*, ECOOP 2010 — Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings (T. D'Hondt, ed.), Lecture Notes in Computer Science, vol. 6183, Springer-Verlag, 2010, pp. 504–528.

[Ehm01] T. Ehm, *Transformational Construction of Correct Pointer Algorithms*, Perspectives of System Informatics (D. Bjørner, M. Broy, and A. V. Zamulin, eds.), Lecture Notes in Computer Science, vol. 2244, Springer-Verlag, 2001, pp. 116–130.

[Ehm03] ———, *The Kleene Algebra of Nested Pointer Structures: Theory and Applications*, Ph.D. thesis, Universität Augsburg, 2003.

[Ehm04] ———, *Pointer Kleene Algebra*, RelMiCS/AKA 2003 (R. Berghammer, B. Möller, and G. Struth, eds.), Lecture Notes in Computer Science, vol. 3051, Springer-Verlag, 2004, pp. 99–111.

[EKMS92] M. Erne, J. Koslowski, A. Melton, and G. E. Strecker, *A Primer on Galois Connections*, York Academy of Science, 1992.

[FBH97] M. F. Frias, G. Baum, and A. M. Haeberer, *Fork Algebras in Algebra, Logic and Computer Science*, Fundam. Inform. **32** (1997), no. 1, 1–25.

[FL79] M. J. Fischer and R. E. Ladner, *Propositional Dynamic Logic of Regular Programs*, Journal of Computer and System Sciences **18** (1979), no. 2, 194–211.

[GGN11] D. Garbervetsky, D. Gorín, and A. Neisen, *Enforcing Structural Invariants using Dynamic Frames*, Proceedings of the 17th Intl Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'11/ETAPS'11, Springer-Verlag, 2011, pp. 65–80.

[GLW06] D. Galmiche and D. Larchey-Wendling, *Expressivity Properties of Boolean BI Through Relational Models*, Proceedings of the 26th International Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS'06, Springer-Verlag, 2006, pp. 357–368.

[HHM+11] C. A. R. Hoare, A. Hussain, B. Möller, P. W. O'Hearn, R. L. Petersen, and G. Struth, *On Locality and the Exchange Law for Concurrent Processes*, CONCUR 2011 (J. P. Katoen and B. König, eds.), Lecture Notes in Computer Science, vol. 6901, Springer-Verlag, 2011, pp. 250–264.

[HMSW09a] C. A. R. Hoare, B. Möller, G. Struth, and I. Wehrman, *Concurrent Kleene Algebra*, CONCUR 09 — Concurrency Theory (M. Bravetti and G. Zavattaro, eds.), Lecture Notes in Computer Science, vol. 5710, Springer-Verlag, 2009, pp. 399–414.

[HMSW09b] ———, *Foundations of Concurrent Kleene Algebra*, Relations and Kleene Algebra in Computer Science (R. Berghammer, A. Jaoua, and B. Möller, eds.), Lecture Notes in Computer Science, vol. 5827, Springer-Verlag, 2009, pp. 166–186.

[HMSW11] C. A. R. Hoare, B. Möller, G. Struth, and I. Wehrman, *Concurrent Kleene Algebra and its Foundations*, Journal of Logic and Algebraic Programming **80** (2011), no. 6, 266–296.

[Hoa69] C. A. R. Hoare, *An Axiomatic Basis for Computer Programming*, Communications of the ACM **12** (1969), no. 10, 576–580, Reprint in [BD02].

[Hoa72] ———, *Proofs of Correctness of Data Representations*, Acta Informatica **1** (1972), 271–281.

[Hoa11] ———, *An Algebra for Program Designs*, Notes on Summer School in Software Engineering and Verification in Moscow, 2011.

[Höf] P. Höfner, *Database for Automated Proofs of Kleene Algabra*, `http://www.dcs.shef.ac.uk/~georg/ka` (accessed December 17, 2014).

[Höf08] ———, *Automated reasoning for hybrid systems — Two case studies*, Relations and Kleene Algebra in Computer Science (R. Berghammer, B. Möller, and G. Struth, eds.), Lecture Notes in Computer Science, vol. 4988, Springer-Verlag, 2008, pp. 191–205.

[Höf09] ———, *Algebraic Calculi for Hybrid Systems*, Ph.D. thesis, Universität Augsburg, 2009.

[HS07] P. Höfner and G. Struth, *Automated Reasoning in Kleene Algebra*, Automated Deduction — CADE-21 (F. Pfenning, ed.), Lecture Notes in Artificial Intelligence, vol. 4603, Springer-Verlag, 2007, pp. 279–294.

[HS08] ———, *Can Refinement be Automated?*, Refine 2007 (E. Boiten, J. Derrick, and G. Smith, eds.), Electronic Notes on Theoretical Computer Science, vol. 201, Elsevier Science Publishers Ltd., 2008, pp. 197–222.

[HSS08] P. Höfner, G. Struth, and G. Sutcliffe, *Automated Verification of Refinement Laws*, Annals of Mathematics and Artificial Intelligence, Special Issue on First-order Theorem Proving **1** (2008), 35–62.

[HV13] A. Hobor and J. Villard, *The Ramifications of Sharing in Data Structures*, Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (R. Giacobazzi and R. Cousot, eds.), POPL, ACM Press, 2013, pp. 523–536.

[IO01] S. Ishtiaq and P. W. O'Hearn, *BI as an Assertion Language for Mutable Data Structures*, ACM SIGPLAN Notices **36** (2001), no. 3, 14–26.

[JB12] J. B. Jensen and L. Birkedal, *Fictional Separation Logic*, Proceedings of the 21st European Conference on Programming Languages and Systems, ESOP'12, Springer-Verlag, 2012, pp. 377–396.

[JP08] B. Jacobs and F. Piessens, *The VeriFast Program Verifier*, Tech. Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven, Belgium, 2008.

[JT51] B. Jónsson and A. Tarski, *Boolean Algebras with Operators, Part I*, American Journal of Mathematics **73** (1951), 891–939.

[Kas11] I. T. Kassios, *The Dynamic Frames Theory*, Formal Aspects of Computing **23** (2011), no. 3, 267–289.

[Koz94] D. Kozen, *A Completeness Theorem for Kleene Algebras and the Algebra of Regular Events*, Information and Computation **110** (1994), no. 2, 366–390.

[Koz97] ———, *Kleene Algebra with Tests*, ACM Transactions on Programming Languages and Systems **19** (1997), no. 3, 427–443.

[Koz00] ———, *On Hoare Logic and Kleene Algebra with Tests*, ACM Transactions on Computational Logic **1** (2000), no. 1, 60–76.

[Koz02] ———, *On Hoare Logic, Kleene Algebra, and Types*, In the Scope of Logic, Methodology, and Philosophy of Science: Volume One of the 11th Int. Congress Logic, Methodology and Philosophy of Science, Cracow, August 1999 (P. Gärdenfors, J Woleński, and K. Kijania-Placek, eds.),

Studies in Epistemology, Logic, Methodology, and Philosophy of Science, vol. 315, Kluwer Academic Publishers, 2002, pp. 119–133.

[KP00] D. Kozen and M.-C. Patron, *Certification of Compiler Optimizations using Kleene Algebra with Tests*, Proceedings of the 1st International Conference on Computational Logic (CL2000) (J. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L. Moniz Pereira, Y. Sagiv, and P. J. Stuckey, eds.), Lecture Notes in Computer Science, vol. 1861, Springer-Verlag, 2000, pp. 568–582.

[Lam68] J. Lambek, *Deductive Systems and Categories I. Syntactic Calculus and Residuated Categories*, Mathematical Systems Theory **2** (1968), no. 4, 287–318.

[Lei10] K. Rustan M. Leino, *Dafny: An Automatic Program Verifier for Functional Correctness*, Proceedings of the 16th Intl. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning, Springer-Verlag, 2010, pp. 348–370.

[Mad06] R. Maddux, *Relation Algebras*, Studies in Logic and the Foundations of Mathematics, vol. 150, Elsevier Science Publishers Ltd., 2006.

[MB85] E. Manes and D. Benson, *The Inverse Semigroup of a Sum-Ordered Semiring*, Semigroup Forum **31** (1985), 129–152.

[McC05] W. McCune, *Prover9 and Mace4*, `http://www.cs.unm.edu/~mccune/prover9`, 2005.

[MH69] J. McCarthy and P. J. Hayes, *Some Philosophical Problems from the Standpoint of Artificial Intelligence*, Machine Intelligence 4 (B. Meltzer and D. Michie, eds.), Edinburgh University Press, 1969, pp. 463–502.

[MHS06] B. Möller, P. Höfner, and G. Struth, *Quantales and Temporal Logics*, Algebraic Methodology and Software Technology (M. Johnson and V. Vene, eds.), Lecture Notes in Computer Science, vol. 4019, Springer-Verlag, 2006, pp. 263–277.

[Möl92] B. Möller, *Some Applications of Pointer Algebra*, Programming and Mathematical Method (M. Broy, ed.), NATO ASI Series, Series F: Computer and Systems Sciences, no. 88, Springer-Verlag, 1992, pp. 123–155.

[Möl93a] _____, *Derivation of Graph and Pointer Algorithms*, Springer-Verlag, 1993.

[Möl93b] _____, *Towards Pointer Algebra*, Science of Computer Programming **21** (1993), no. 1, 57–90.

[Möl97] _____, *Calculating with Pointer Structures*, Proceedings of the IFIP TC2/WG 2.1 International Workshop on Algorithmic Languages and Calculi, Chapman & Hall, 1997, pp. 24–48.

[Möl99a] _____, *Calculating with Acyclic and Cyclic Lists*, Information Sciences **119** (1999), no. 3-4, 135–154.

[Möl99b] _____, *Calculational System Design*, Nato Science Series, Series F: Computer and System Sciences, vol. 173, ch. Algebraic Structures for Program Calculation, pp. 25–97, IOS Press, 1999.

[Möl07] _____, *Kleene Getting Lazy*, Science of Computer Programming **65** (2007), 195–214.

[MS06a] B. Möller and G. Struth, *Algebras of Modal Operators and Partial Correctness*, Theoretical Computer Science **351** (2006), no. 2, 221–239.

[MS06b] _____, *wp Is wlp*, Relational Methods in Computer Science (W. MacCaull, M. Winter, and I. Düntsch, eds.), Lecture Notes in Computer Science, vol. 3929, Springer-Verlag, 2006, pp. 200–211.

[Mul86] C. Mulvey, &, Rendiconti del Circolo Matematico di Palermo **12** (1986), no. 2, 99–104.

[Ngu91] T. T. Nguyen, *A Relational Model of Nondeterministic Programs*, International Journal on Foundations of Computer Science **2** (1991), 101–131.

[Nis06] S. Nishimura, *Reasoning About Data-Parallel Pointer Programs in a Modal Extension of separation logic*, Algebraic Methodology and Software Technology, 11th International Conference (M. Johnson and V. Vene, eds.), Lecture Notes in Computer Science, vol. 4019, Springer-Verlag, 2006, pp. 293–307.

[O'H07] P. W. O'Hearn, *Resources, Concurrency, and Local Reasoning*, Theoretical Computer Science **375** (2007), no. 1–3, 271–307.

[OP99] P. W. O'Hearn and D. J. Pym, *The Logic of Bunched Implications*, Bulletin of Symbolic Logic **5** (1999), no. 2, 215–244.

[ORY01] P. W. O'Hearn, J. C. Reynolds, and H. Yang, *Local Reasoning about Programs that Alter Data Structures*, CSL '01: Proceedings of the 15th

International Workshop on Computer Science Logic (L. Fribourg, ed.), Lecture Notes in Computer Science, vol. 2142, Springer-Verlag, 2001, pp. 1–19.

[ORY09] ———, *Separation and Information Hiding*, ACM Transactions on Programming Languages and Systems **31** (2009), no. 3, 1–50.

[Par10] M. Parkinson, *The Next 700 Separation Logics*, Verified Software: Theories, Tools, Experiments (G. T. Leavens, P. W. O'Hearn, and S. K. Rajamani, eds.), Lecture Notes in Computer Science, vol. 6217, Springer-Verlag, 2010, pp. 169–182.

[PB05] M. Parkinson and G. Bierman, *Separation Logic and Abstraction*, Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '05, ACM Press, 2005, pp. 247–258.

[PBC06] M. Parkinson, R. Bornat, and C. Calcagno, *Variables as Resource in Hoare Logics*, Proceedings of the Twenty-First Annual IEEE Symposium on Logic in Computer Science (LICS 2006), IEEE Computer Society Press, 2006, pp. 137–146.

[PBO07] M. Parkinson, R. Bornat, and P. O'Hearn, *Modular Verification of a Non-blocking Stack*, Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '07, ACM Press, 2007, pp. 297–302.

[Plo04] G. D. Plotkin, *A Structural Approach to Operational Semantics*, Journal of Logic and Algebraic Programming **60–61** (2004), 17–139.

[POY04] D. J. Pym, P. W. O'Hearn, and H. Yang, *Possible Worlds and Resources: The Semantics of BI*, Theoretical Computer Science **315** (2004), no. 1, 257–305.

[PPS10] D. Pavlovic, P. Pepper, and D. R. Smith, *Formal Derivation of Concurrent Garbage Collectors*, Mathematics of Program Construction (C. Bolduc, J. Desharnais, and B. Ktari, eds.), Lecture Notes in Computer Science, vol. 6120, Springer-Verlag, 2010, pp. 353–376.

[Pre09] V. Preoteasa, *Frame Rule for Mutually Recursive Procedures Manipulating Pointers*, Theoretical Computer Science **410** (2009), no. 42, 4216–4233.

[PS11] M. Parkinson and A. J. Summers, *The Relationship Between Separation Logic and Implicit Dynamic Frames*, Proceedings of the 20th European Conference on Programming Languages and Systems: Part of the Joint European Conferences on Theory and Practice of Software, ESOP'11/ETAPS'11, Springer-Verlag, 2011, pp. 439–458.

[PS12] _____, *The Relationship Between Separation Logic and Implicit Dynamic Frames*, Logical Methods in Computer Science **8** (2012), no. 3, 1–54.

[Pym02] D. J. Pym, *The Semantics and Proof Theory of the Logic of Bunched Implications*, Applied Logic Series, no. 26, Kluwer Academic Publishers, 2002, Errata and remarks (Pym 2008) maintained at http://homepages.abdn.ac.uk/d.j.pym/pages/BI-monograph-errata.pdf.

[Rey00] J. C. Reynolds, *Intuitionistic Reasoning about Shared Mutable Data Structure*, Millennial Perspectives in Computer Science (J. Davies, B. Roscoe, and J. Woodcock, eds.), Palgrave, 2000, pp. 303–321.

[Rey02] _____, *Separation Logic: A Logic for Shared Mutable Data Structures*, LICS '02: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science, IEEE Computer Society, 2002, pp. 55–74.

[Rey08] _____, *Lecture Notes for the First Phd Fall School on Logics and Semantics of state*, 2008, http://www.cs.cmu.edu/~jcr/copenhagen08.pdf.

[Rey09] _____, *An Introduction to Separation Logic*, In Engineering Methods and Tools for Software Safety and Security (M. Broy, ed.), IOS Press, 2009, pp. 285–310.

[RG08] M. Raza and P. Gardner, *Footprints in Local Reasoning*, Foundations of Software Science and Computational Structures (R. Amadio, ed.), Lecture Notes in Computer Science, vol. 4962, Springer-Verlag, 2008, pp. 201–215.

[Ros90] K. I. Rosenthal, *Quantales and their Applications*, Pitman Research Notes in Mathematics Series, vol. 234, Longman Scientific & Technical, 1990.

[Sim06] E.-J. Sims, *Extending Separation Logic with Fixpoints and Postponed Substitution*, Theoretical Computer Science **351** (2006), no. 2, 258–275.

[SJP09] J. Smans, B. Jacobs, and F. Piessens, *Implicit Dynamic Frames: Combining Dynamic Frames and Separation Logic*, Proc of the 23rd European Conf. on ECOOP, Genoa, Springer-Verlag, 2009, pp. 148–172.

[SS93] G. Schmidt and T. Ströhlein, *Relations and Graphs: Discrete Mathematics for Computer Scientists*, Springer-Verlag, 1993.

[SS98] G. Sutcliffe and C.B. Suttner, *The TPTP Problem Library: CNF release v1.2.1*, Journal of Automated Reasoning **21** (1998), no. 2, 177–203.

[Str07] G. Struth, *Reasoning Automatically about Termination and Refinement*, 6th International Workshop on First-Order Theorem Proving (S. Ranise, ed.), vol. Technical Report ULCS-07-018, Department of Computer Science, University of Liverpool, 2007, pp. 36–51.

[SW67] H. Schorr and W. M. Waite, *An Efficient Machine-independent Procedure for Garbage Collection in Various List Structures*, Communications of the ACM **10** (1967), no. 8, 501–506.

[Tar55] A. Tarski, *A Lattice-theoretical Fixpoint Theorem and its Applications*, Pacific Journal of Mathematics **5** (1955), no. 2, 285–309.

[TBY12] J. Thamsborg, L. Birkedal, and H. Yang, *Two for the Price of One: Lifting Separation Logic Assertions*, Logical Methods in Computer Science **8** (2012), no. 3:22, 1–31.

[TKN07] H. Tuch, G. Klein, and M. Norrish, *Types, Bytes, and Separation logic*, POPL '07: Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Press, 2007, pp. 97–108.

[TSBR08] N. Torp-Smith, L. Birkedal, and J. C. Reynolds, *Local Reasoning about a Copying Garbage Collector*, ACM Transactions on Programming Languages and Systems **30** (2008), no. 4, 24:1–24:58.

[Tue08] T. Tuerk, *A Separation Logic Framework in HOL*, Tech. Report 2008-1-Ait Mohamed, Department of Electrical Computer Engineering, Concordia University, 2008.

[Vaf11] V. Vafeiadis, *Concurrent Separation Logic and Operational Semantics*, Electronic Notes on Theoretical Computer Science **276** (2011), 335–351.

[VK98] B. Von Karger, *Temporal Algebra*, Mathematical Structures in Computer Science **8** (1998), no. 3, 277–320.

[VP07] V. Vafeiadis and M. J. Parkinson, *A Marriage of Rely/Guarantee and Separation logic*, CONCUR 2007 - Concurrency Theory, 18th International Conference, Lecture Notes in Computer Science, vol. 4703, Springer-Verlag, 2007, pp. 256–271.

[WBO08] S. Wang, L.-S. Barbosa, and J.-N. Oliveira, *A Relational Model for Confined Separation Logic*, Proc. of the 2nd IFIP/IEEE Intl. Symposium on Theoretical Aspects of Software Engineering, TASE '08, IEEE Computer Society Press, 2008, pp. 263–270.

[WHO09] I. Wehrman, C. A. R. Hoare, and P. W. O'Hearn, *Graphical Models of Separation Logic*, Information Processing Letters **109** (2009), no. 17, 1001–1004.

[Win07] M. Winter, *Goguen Categories: A Categorical Approach to L-fuzzy Relations*, 1st ed., Springer Publishing Company, Incorporated, 2007.

[Yan01] H. Yang, *An Example of Local Reasoning in BI Pointer Logic: The Schorr-Waite Graph Marking Algorithm*, SPACE 2001: Informal proceedings of Workshop on Semantics, Program Analysis and Computing Environments for Memory Management (F. Henglein, J. Hughes, H. Makholm, and H. Niss, eds.), 2001, pp. 41–68.

[Yan07] ———, *Relational Separation Logic*, Theoretical Computer Science **375** (2007), no. 1–3, 308–334.

[YLB+08] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. O'Hearn, *Scalable Shape Analysis for Systems Code*, Computer Aided Verification, 20th International Conference (A. Gupta and S. Malik, eds.), Lecture Notes in Computer Science, vol. 5123, Springer-Verlag, 2008, pp. 385–398.

[YO02] H. Yang and P. W. O'Hearn, *A Semantic Basis for Local Reasoning*, Foundations of Software Science and Computation Structures, Proceedings FOSSACS 2002 (M. Nielsen and U. Engberg, eds.), Lecture Notes in Computer Science, vol. 2303, Springer-Verlag, 2002, pp. 402–416.

List of Figures

2.1 Illustration of separating implication. 15
2.2 Examples of Hoare triples in separation logic. 19
2.3 Hoare logic inference rules. 19

3.1 Notations of operators in separation logic, AS and abstract quantales. 30

4.1 Relational semantics of heap-independent commands. 59
4.2 Relational semantics of heap-dependent commands. 60
4.3 Recursive definition of interleaving traces. 88
4.4 Dependencies in the exchange law. 100
4.5 Compatibility in the reverse exchange law. 103
4.6 Illustration of the cross-split assumption for a state σ. 111
4.7 Specification of a rational number module with dynamic frames. . . . 112
4.8 State partitions of a state σ for a bounded frame g. 118

5.1 Sharing within two singly linked lists. 124
5.2 Examples of sharing patterns for addresses x_1, x_2, x_3. 127
5.3 Illustration of $\triangleright$ on trees. 134
5.4 $\triangleright$-combination of forests a, b. 136
5.5 Illustration of a selector assignment inference rule. 144
5.6 Verification of list reversal. 149
5.7 Tree rotation at the beginning and end. 150
5.8 Verification of tree rotation. 151
5.9 A shared subtree. 152
5.10 Tree rotation with sharing. 153
5.11 Depiction of the intermediate state with sharing. 153
5.12 A threaded tree. 154
5.13 Verification of adding a new node to a threaded tree. 158

A.1 Definition of the free variables FV(_) of syntactical commands. 184
A.2 Definition of the modified variables MV(_) of syntactical commands. . 184

Index

$*$-product, *see* separating conjunction

abstraction function, 148
access element, 127
    acyclic, 132
    closed, 131
    deterministic, 133
assertions, 14
    fully allocated, 43
    intuitionistic, 33
    precise, 40
    pure, 36
    supported, 44

bunched implications (**BI**), 31

codomain, 125
commands, 16
    relational interpretation, 58
compatible, 102
    backward, 102
    forward, 101
concurrency rule, 86
concurrent Kleene algebra (**CKA**), 98

detachment, 29
directed combinability, 134
domain
    abstract, 125
    relations, 62
dynamic frames, 109
    accumulation, 119
    bounded, 117

exchange law, 99
    reverse, 102

frame property, 21
    generalised, 75
    pointfree, 72
    relational, 73
frame rule, 20
    generalised, 80
framing requirements, 113
    modification, 113
    preservation, 113

Hoare triple
    resource-sensitive, 63
    concurrent separation logic, 92
    general, 100
    partial correctness, 66
    pointwise, 18
    total correctness, 67

iteration, 125

Kleene algebra, 125

linked structure, 133
    cell, 134
    chain, 134
    forest, 133
    root, 134

tree, 134
local actions, 81, 82
locality, 116
bimonoid, 109

modal operators
abstract, 125
relations, 64

operational semantics, 17

parallel decomposition, 89
pointfree, 90
permission algebra, 53
counting permissions, 54
fractional permissions, 53
predicate transformer, 83
locality, 83
preservation, 76
strong, 91
properness, 126

quantale, 24

relation
angelic behaviour, 60
Cartesian product, 69
compensator, 74
demonic behaviour, 65
domain-precise, 97
join, 68
split, 68
residual, 27
resource context, 86
respect program abortion, 63

safe states, 64
safety monotonicity, 21
selector, 133
separating conjunction, 14
AS, 24
separating implication, 14
AS, 27
separation algebra, 51
separation logic, 12
concurrent, 85
transitive, 123
septraction, 29
strong disjointness, 127

termination monotonicity, 80
tests, 39
atomic, 126
pairs, 69
threaded trees, 154
twig, 141
update, 141

Curriculum Vitae

Han Hing Dang

Personal Data	
Date of birth	February 19, 1984
Citizenship	German
Education	
since May 2009	Doctorate, University of Augsburg, Germany
March 2009	Diploma in Computer Science (major) and in Mathematics (minor)
October 2004 – March 2009	Study of Computer Science
July 2004	Abitur
Academic Positions	
since October 2010	Researcher, University of Augsburg, Germany
May 2009 – September 2010	Research Assistant, University of Augsburg, Germany